目录

第一章　Object Pascal 语法新特点 …… 1

实现动态数组 …… 2
动态创建对象 …… 5
实现过程的可变参数调用 …… 9

第二章　窗体设计 …… 17

创建无窗口应用程序 …… 18
显示启动封面 …… 19
实现无标题窗体的移动 …… 22
捕获鼠标在窗体中非客户区的消息 …… 24
防止用户更改窗体位置和大小 …… 28
在任务栏中隐藏应用程序 …… 31
防止关闭 Windows …… 32

第三章　MDI 应用 …… 35

MDI 应用程序是怎样构成的 …… 36
为 MDI 主窗体增加背景 …… 41
隐藏子窗体 …… 45
合并 MDI 菜单 …… 50

第四章　菜单 …… 53

添加系统菜单项 …… 54
为菜单项增加提示功能 …… 57
截获菜单的非加速键 …… 59
动态创建菜单项 …… 61
在菜单中动态关联过程 …… 66

第五章　控件的使用 …… 71

在运行时用鼠标移动控件 …… 72
在 ListBox 和 ComboBox 中插入图片 …… 74
为 ListBox 加入水平滚动条 …… 80

截获滚动条的滚动消息 …… 82
动态创建组件 …… 86

第六章　资源文件 …… 93

自定义和使用资源文件 …… 94
从资源文件中装载位图和光标 …… 98
将 wav 文件包含到应用程序中 …… 101
使用其他 DLL 或 EXE 文件中的资源文件 …… 104

第七章　文件、目录及驱动器 …… 107

获得可用驱动器及其类型 …… 108
获取驱动器信息 …… 111
获取几种常用目录 …… 118
查找文件 …… 123

第八章　剪贴板 …… 129

复制、粘贴文本和图形数据 …… 130
自动感知剪贴板操作 …… 135
创建自定义剪贴板格式 …… 139

第九章　串口通信 …… 149

通过串口配置对话框更改串口配置 …… 150
实现串口通信 …… 156

第十章　打印机与打印 …… 161

指定当前要使用的打印机 …… 162
获取和设置打印机基本信息 …… 165
控制打印机设置 …… 170
获取打印机详细信息 …… 178

第十一章　Delphi 消息系统 …… 185

理解 Delphi 的消息系统 …… 186
截获组件消息 …… 194
自定义消息 …… 199

第十二章　Windows 系统控制 …… 205

创建控制面板小应用程序 …… 206
调用控制面板设置功能 …… 212
检测即插即用硬件发生的变化 …… 218

直接控制 I/O …… 221
获取系统一般信息 …… 224
检查环境变量 …… 234
设置桌面墙纸 …… 236
捕获系统按键 …… 238

第十三章　API 技术 …… 243

防止加载应用程序的多个例程 …… 244
获得另一程序的窗体句柄 …… 248
列举指定窗体的子窗体 …… 253
将应用程序图标放到托盘中 …… 259
创建应用程序工具栏 …… 268

第十四章　图形与图像 …… 281

理解 Windows GDI 与 TCanvas …… 282
直接在控件上绘图 …… 285
将透明位图画到桌面上 …… 291
截取屏幕图像 …… 295
自定义字体 …… 297

第十五章　多媒体编程 …… 301

控制光驱 …… 302
设置光驱自动播放功能 …… 307
获取 Audio CD 信息 …… 310
捕获视频图像 …… 315
制作屏幕保护程序 …… 321
打开和关闭屏幕保护程序 …… 340

第十六章　DBGrid 高级应用 …… 343

实现 DBGrid 的下拉列表框输入 …… 344
为 DBGrid 插入组件 …… 347
改变 DBGrid 单元格缺省的颜色 …… 350
在 DBGrid 中实现拖拉技术 …… 354

第十七章　快速报表 …… 359

快速创建报表 …… 360
创建自定义预览 …… 372
进行 QuickReport 附件的开发 …… 377

第十八章　BDE 应用 …… 383

在代码中创建 BDE 别名 …… 384
控制 BDE 别名的信息 …… 386

第十九章　数据库 …… 391

自动登录数据库服务器 …… 392
动态创建数据库表格 …… 393
将文件存入数据库中 …… 398
实现计算字段 …… 405

第二十章　MIDAS 技术 …… 411

认识 MIDAS 技术的特点 …… 412
认识 MIDAS 技术 …… 413
创建应用服务器 …… 419
创建客户端应用 …… 431
定制应用服务器 …… 438
创建 ActiveX 控件分发客户应用程序 …… 440

第二十一章　帮助文件 …… 443

理解帮助系统的组成 …… 444
进行帮助系统的设计 …… 447
建立帮助系统的主题结构 …… 448
设计帮助项目文件和目录文件 …… 454
调用帮助文件 …… 459

第二十二章　安装盘制作 …… 463

制作安装盘 …… 464

第一章

Object Pascal 语法新特点

热点透视

Delphi4 和 Delphi5 中对 Object Pascal 语言本身也做了一些改进，增加了对动态数组、函数重载、默认参数、64 位整数、32 位无符号整数、改变了的实数类型以及 Implements 接口等支持。由于动态数组、函数重载以及缺省参数的使用对程序员的编程方式、风格甚至算法等都会产生影响，程序员实现一些程序变得更加容易，并且使程序的可读性更强，因此，本章将针对这些方面进行介绍。另外，还介绍了如何动态创建对象。



实现动态数组



遇到难题

相信不少编程人员对 Pascal 中的数组已经相当熟悉了，但在 Delphi3 之前，有的用户还也许没有使用过动态数组，尽管在 Pascal 中同 C 一样可以通过指针来使用动态数据。以前，为了实现动态数组，通常由指针变量来定义一条链，这样操作起来非常繁琐，同时还得让每一个元素必须记录前后元素的地址。现在，在 Delphi4 和 Delphi5 中已经增加了动态数组这一数据类型。那么，在新版的 Delphi 中又如何使用动态数组呢？



钥匙在此

跟使用静态数组相类似，为了使用动态数组，必须经过变量的声明和变量的使用这两个过程。但是在使用之前必须指定数组的大小以及给数组分配空间。

1. 动态数组声明

在 Delphi4 或 Delphi5 中，可以声明静态数组，其声明格式如下：

```
A: array[1..100] of string;
```

现在还可以声明动态数组。动态数组的维数和长度在编译期间是不确定的。动态数组只指定类型信息（如数组的维数以及元素的类型）而不指定元素数。

动态数组并没有固定的长度或大小。不过，动态数组的内存分配会在给它赋值时或将其作为参数传给过程函数 SetLength 时进行分配。动态数组的类型的声明格式为：

```
array of baseType
```

例如，

```
var B: array of string;
```

2. 为动态数组指定大小并分配内存空间

上例中声明了一维字符串动态数组 B，但还没有为 B 分配内存。为了给数组分配内存空间，调用 SetLength 过程函数。例如，在上述声明基础上，调用

```
SetLength(B, 20);
```

分配了包含 20 个字符串类型元素的数组，下标索引为 0~19。分配了内存后，就可以像访问一般的数组那样访问动态数组的每一个元素了：

```
B[0]:='我是一个字符串！';
showmessage(B[0]);
```



动态数组是以整数作为下标的，而且总是从 0 开始。

动态数组也可以是多维的。若要指定多维动态数组，可以参考下面的代码：

```
var
    //基类型为整数的二维动态数组
    IA:array of array of integer;
```

要为多维的动态数组分配内存，可以参考下面的代码：

```
    //IA 将是 5X5 的整型数组
    SetLength(IA,5,5);
```

访问多维的动态数组就像访问一般的多维数组一样，如：

```
    IA[0,3]:=2;
```



在一些函数或过程声明中，数组参数是以 array of baseType 声明的，没有指定任何下标索引，例如：

```
function CheckStrings(A: array of string): Boolean;
```

这就意味着该函数主要关心数组的基类型，而不是数组的大小、索引方法或者是动态还是静态的方式分配内存等方面内容。

3. 为动态数组释放空间

动态数组具有生存期自管理功能。因此，没有必要专门去释放它所占有的内存。不过，如果想在动态数组超出其作用域之前删除它（如果它占用了很多内存的话），则只要用 Nil 对动态数组赋值即可。程序示例如下：

```
B:=Nil;//释放数组空间
```

4. 动态数组相互赋值

与字符串类似，动态数组相互赋值时，赋值的仅仅是对数据的引用，而非数据本身，这一点与一般的数组不同。请看下面的代码：

```
var
  A, B: array of Integer;
begin
  SetLength(A, 4);//为数组 A 分配空间
  A[0] := 1;//为元素 0 赋值
  B := A;
  B[0] := 2;
end;
```

A[0]现在等于几呢？正确答案为 2。这是因为赋值语句 B:=A 并没有为 B 单独分配内存，而是使 B 与 A 引用同一数组。这样，任何对 B 的修改必然会影响到 A。如果想把 A 的数据复制一个副本给 B，就应当调用标准过程 Copy()：

```
B:=Copy(A);
```

上面这行代码执行后，B 与 A 就成为两个独立的数组，尽管它们的初始值是相同的，但是修改其中一个数组的数据不会影响另一个数组。另外，调用 Copy()时，还可以指定从

第几个元素开始复制、复制几个元素等，示例如下：

```
//从第 1 个元素开始复制 2 个元素
B:=Copy(A,1,2);
```

下面来看一下创建多维动态数组的完整例子。



跟我来

为了声明多维动态数组，可以重复使用 array of ...结构，如：

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

这样就声明了一个二维字符串动态数组。如果 I,J 是两个整数型变量，则

```
SetLength(Msgs,I,J);
```

表示分配了大小为 I×J 的数组，并且 Msgs[0,0]表示该数组的第 1 个元素。

另外，也可以使用另一种方式来为动态数组分配内存。如：

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

可以为 Ints 分配 10 行空间。然后，还可以单独为每一行分配空间。例如：

```
SetLength(Ints[2], 5);
```

为第 3 行分配了 5 个整数空间。此时，还可以给该行中的元素进行赋值，虽然其他的行还未分配列。如 Ints[2,4] := 63。

下面的例子使用了动态数组来创建三角矩阵：

```
Var
  A : array of array of string;
  I, J : Integer;
  s:string;
Begin
  //为数组 A 分配 10 行空间
  SetLength(A, 10);
  //I 变量从 A 的最小下标 0 到 A 的最大下标 9 进行循环
  for I := Low(A) to High(A) do
  begin
    //为 A[I]行分配 I 列空间
    SetLength(A[I], I);
    //J 变量从 A[I]的最小下标到 A[I]的最大下标进行循环
    for J := Low(A[I]) to High(A[I]) do begin
      A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
      s:=s+A[I,j];
        end;
        s:=s+#13;
```

```
    end;
    showmessage(s);
end;
```

运行该段程序，结果如图 1-1 所示。

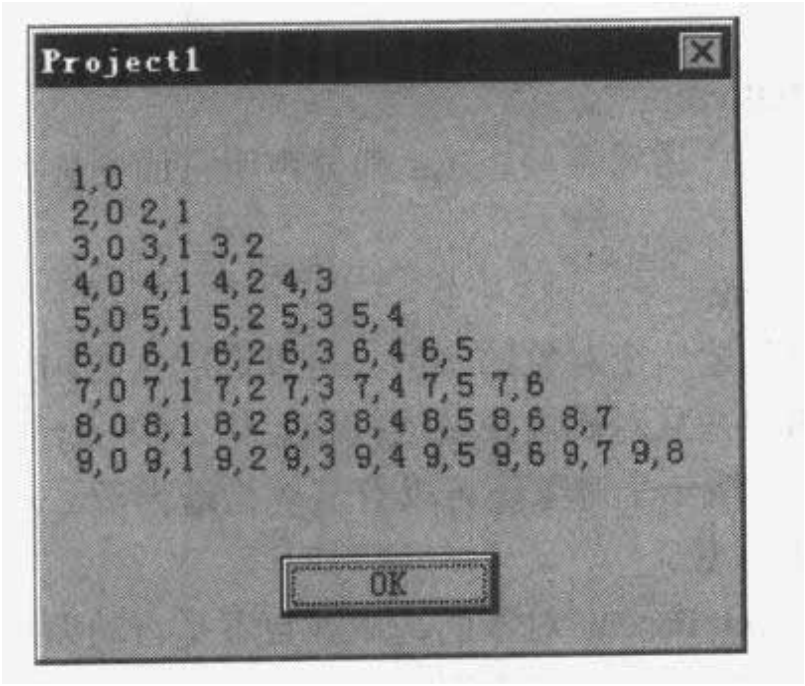


图 1-1　三角矩阵



Low()和 High()两个函数分别返回一个范围内的最小值和最大值。如果该函数的参数为数组的话，就将返回该数组的下标最小值和下标最大值。



动态创建对象



遇到难题

在使用 Delphi 编程时，一般在设计阶段就把整个界面都制作完毕，但有时也会碰到在 Delphi 的应用程序运行时，需要创建一些界面元素的情况，如在某些条件下，需要动态地为窗体创建一个按钮，一个文本框，甚至需要动态地创建一些窗体等。编一个不同风格的界面时更是如此。那么，在程序执行时如何动态地创建这些组件或者其他对象呢？



钥匙在此

在 OOP（面向对象编程）技术中，对象是同时包含数据和操作的实体，它是类的一个实例（OOP 是一项非常成熟的技术，并且有很多资料都详细介绍过这方面的内容，因此，本书将不再赘述）。Delphi 的对象具有面向对象编程所具有的强大优势，全面支持继承、封

装和多态。

1. 声明和创建实例

在使用一个对象之前，首先必须使用关键字 class 来声明该对象的类。对象应当在程序单元文件的 type 部分进行声明，如：

```
type
TmyButton=class(TButton);
```

声明了一种对象类型后，通常还要在 var 部分声明它的变量或实例：

```
var
    myButton:TmyButton;
```

在 Object Pascal 中要创建一个对象的实例，可以调用它的构造函数。构造函数的作用是创建对象的实例、分配内存并且对属性或域进行初始化。Object Pascal 的对象至少要有一个叫 Create()的构造函数，当然一个对象还可以有其他构造函数。对于不同类型的对象来说，Create()可以带不同数量的参数。

与 C++不同的是 Object Pascal 对象的构造函数不是自动调用的，程序员必须自己调用构造函数。调用构造函数的文法可以参考下面的例子：

```
myButton:=TmyButton.Create(Nil);
```

调用构造函数的文法有点特殊。Create()是由类型调用的而不是实例调用的，而其他方法一般只能由实例来调用。通过调用构造函数来创建对象的实例，这就是所谓的实例化。



通过调用构造函数创建了对象的实例后，编译器保证对象的每一个域或属性都已经初始化。这时就可以放心地认为所有的数字都初始化为 0，所有的指针都初始化为 Nil，所有的字符串都初始化为空字符串。

2. 析构函数

当对象使用完毕时，应当及时地调用它的 Free()方法来删除它。Free()方法首先检查对象实例是否为 Nil，如果不是的话，就调用对象的析构函数即 Destroy()。析构与构造的作用恰好相反，它释放先前分配的内存，并做一些清除工作，以保证对象从内存中彻底清除。调用析构的文法请参考下面的例子：

```
myButton.Free;
```

与调用 Create()不同的是，必须通过对象实例来调用 Free()。记住，不要直接调用 Destroy()，而要调用 Free()。

这些方法是怎样放在一个小小的对象中的呢？实际上，这些方法都来自于 Object Pascal 的 Tobject 对象，在 Object Pascal 语言中，所有的对象都是从 Tobject 继承下来的，不管它们在声明的时候有没有列出 Tobject。例如：

```
Type   TjxgObject=Class;
```

相当于

```
Type TjxgObject=class(TObject);
```

下面看一下在窗体中动态创建按钮的例子。



跟我来

新建一个 Delphi 应用程序，清单 1-1 列出了完整的单元。其中，设置窗体名为 FrmMain，相应的单元文件名为 Main.pas，并在 FrmMain 中添加了一个 MyOnClick()过程，同时在窗体的 OnShow 和 OnClose 事件中添加了代码。

清单 1-1　动态创建按钮

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFrmMain = class(TForm)
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure   MyOnClick(Sender: TObject);
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}
var
    //声明对象实例
    myButton:TButton;

//自定义过程
procedure   TFrmmain.MyOnClick (Sender: TObject);
begin
      showmessage('我是被动态创建的！ ');
```

```
end;

procedure TFrmMain.FormShow(Sender: TObject);
begin
    //创建按钮实例
    myButton:=TButton.Create(self);
  //设置动态按钮的属性
    mybutton.left:=20;
    mybutton.Top:=20;
    myButton.Width:=120;
    myButton.Caption:='动态创建的按钮';
    myButton.Parent:=Self;//该语句使按钮在窗体上可见
    //将 MyOnClick 过程赋值给 OnClick 属性（事件）
    myButton.OnClick:=MyOnClick;

end;

procedure TFrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
     //释放实例
      myButton.Free;
end;

end.
```

运行该程序代码，将出现如图 1-2 所示的窗体，即在设计状态为空白的窗体上自动添加了一个按钮，其标题为“动态创建的按钮”，当单击该按钮时将会弹出一个“我是被动态创建的!”对话框，因为已经将该按钮的单击事件赋给了自定义的一个过程 MyOnClick，所以单击鼠标时自动激发了 MyOnClick 过程。

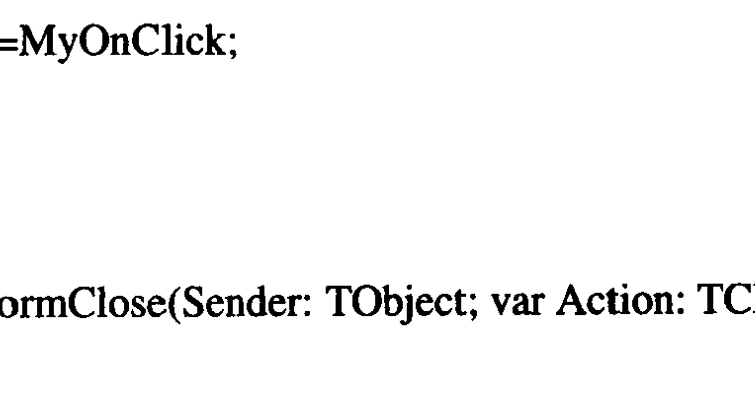


图 1-2　动态创建按钮



关于赋值给对象事件属性的过程或函数的定义，应当符合该事件属性的过程或函数的声明。如，按钮的 OnClick 事件的类型为：

```
type TNotifyEvent = procedure (Sender: TObject) of object;
```

因此，MyOnClick 过程也应该是这种类型。注意在赋值时不能带有参数。

说 明

关于 Self

在任何一个方法中，都有一个隐含的叫 Self 的变量。Self 是一个指针，指向调用这个方法的对象实例。它是由编译器隐含在方法中的。



实现过程的可变参数调用



遇到难题

如果程序员原先只使用一个整型变量作为参数，过一段时间，就会逐渐发现如果设计一个同时使用整型变量和字符串变量作为参数的方法，就可以提高该对象的性能。例如，用户可能会希望通过分别给该方法传递一个整数和机器的 IP 地址来支持分布式计算。但是，在 Delphi3 中，只能通过增加将机器的 IP 地址作为参数的新函数，或者改变原先的函数的参数类型，这两种方法都会使整个程序改动较大，可能会打乱所有的代码，假如是一个大型项目的话，这种情况往往是不允许发生的。那么，在 Delphi4 和 Delphi5 中是不是有一个较好的解决方法呢？答案是肯定的。



钥匙在此

1. 函数的重载

Delphi4 或 Delphi5 中使用函数或方法的重载解决上述问题。可以在 Delphi4 或 Delphi5 中在同样的作用范围内使用相同的函数名声明多个例程。这就是函数重载。这在 C++中早已使用了。

函数的重载可以让用户在程序中使用名字相同而参数不同的函数。重载函数必须以 overload 指令声明，并且这些函数间的参数应当有所区别。例如：

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;
function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
```

```
end;
```

由于这两个函数使用不同的参数，因此，编译器可以区分它们。同时这种能力提高了程序的易维护性，并可以让程序代码更加完美，更加容易读懂。

2. 重载函数的使用

重载函数的使用与一般函数的使用方式一样，只是需要确定一下，什么情况下调用其中哪一个函数而已。



跟我来

现在可以看两个关于函数重载的例子。其中，第一个例子是在同一个类中定义重载函数或方法，第二个例子是对祖先类中的方法进行重载。

1. 同类中函数重载

程序清单 1-2 提供了一个叫作 TmyObject 的类的简单范例，该类支持函数重载。

清单 1-2　一个支持函数重载的简单类

```
unit Main;

interface

uses
  Windows, Messages, SysUtils,
  Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TMyObject = class(TObject)
    function Foo(i: Integer): string; overload;
    function Foo(S: string): string; overload;
  end;

  TForm1 = class(TForm)
    Button1: TButton;
    ListBox1: TListBox;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

function Sam(s: string): Byte; overload;
implementation

{$R *.DFM}

{ TMyObject }

function Sam(s: string): Byte; overload;
begin
  Result := 1;
end;

function Sam(I : Integer): Byte; overload;
begin
  Result := 1;
end;

function TMyObject.Foo(i: Integer): string;
begin
  Result := 'Foo 的整数参数为: ' + IntToStr(i);
end;

function TMyObject.Foo(S: string): string;
begin
  Result := 'Foo 的字符串参数为: ' + S;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  MyObject: TMyObject;
  S: string;
begin
  MyObject := TMyObject.Create;

  S := MyObject.Foo('Sam');
  ListBox1.Items.Add(S);
```

```
    S := MyObject.Foo(1);
    ListBox1.Items.Add(S);

    Sam(2);

    MyObject.Free;
end;

end.
```

这个程序支持一个简单的窗口，窗口中有一个按钮和一个列表框。如果按一下按钮，那么在列表框中将会弹出两条消息：

Foo 的字符串参数为：sam

Foo 的整数参数为：1

本例的代码中最关键的部分是对 TmyObject 类的声明：

```
  TMyObject = class(TObject)
    function Foo(i: Integer): string; overload;
    function Foo(S: string): string; overload;
  end;
```

可以看到，这个类有两个叫作 Foo 的方法。在过去的 Delphi3 中，要拥有这样两个方法是不可能的，当给这两个方法定义了同样的名称时，编译器就会指出有误。而在 Delphi4 和 Delphi5 中，只要每一个方法的参数是不一样的，并且使用了 overload 关键字来区分出要重载的函数，那么这种用法是可行的。

2. *类方法的重载与继承*

类方法的重载可以使祖先类和派生类结构中的方法都能如您所愿地进行工作。换句话说，如果在祖先类中使用 overload 关键字声明了一个方法，那么当在后继的类中使用相同的名称声明重载方法时就不会引起混乱。程序清单 1-3 显示了重载方法在类继承的结构中是如何工作的。

清单 1-3　类继承结构中的函数（方法）重载例子

```
unit Main;

interface

uses
  Windows, Messages, SysUtils,
  Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
```

```
//祖先类 TMyObject
TMyObject = class(TObject)
private
  FName: string;
public
  procedure InitObject(AName: string); overload;      //方法重载声明
end;

//继承类 TAnimal
TAnimal = class(TMyObject)
private
  FType: string;
  public
    procedure InitObject(AName: string; AType: string); overload;
  end;

  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TMyObject.InitObject(AName: string);
begin
  FName := AName;
  ShowMessage('调用了 TMyObject 类');
end;

procedure TAnimal.InitObject(AName: string; AType: string);
begin
  FName := AName;
  FType := AType;
  ShowMessage('调用了 TMyObject 类的子类——TAnimal');
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
var
   A: TAnimal;
begin
   A := TAnimal.Create;
   //重载函数（方法）的调用
   A.InitObject('Object'); //调用了祖先类的方法
   A.InitObject('Harry', 'Horse');//调用了本类的方法
end;

end.
```

这个程序显示了一组跨越对象层次结构的重载方法。在这个例子中，InitObject 是一个重载方法。如果没有使用 overload 这个关键字，那么 InitObject 的第二个实例就会遮蔽住第一个实例。由于代码中使用了 overload，因此在一个单独的 Tanimal 实例中，可以根据需要自由调用这两个方法中的任何一个。



再送礼包

函数的缺省参数

缺省参数可以是给函数参数指定的一个缺省值。请看下面的程序段：

```
procedure   Tform1.Sam(A:integer=1);
begin
      showmessage(inttostr(A));
end;

      procedure   Tform1.Button1Click(sender:Tobject);
         begin
            Sam;
      end;
```

在该例子中，Sam 方法只有一个整型参数。缺省情况下，这个参数将被设置为 3。这就意味着运行上述代码并单击按钮 Button1 时，程序会弹出消息框显示 3 这个值。如果像下面这样调用的话，Sam 将会弹出消息框显示 1 这个值。

```
Sam(1);
```

这和允许函数采用可变数目的参数不是一回事。Sam 过程总是使用一个参数，只不过有时用的是缺省值。

Object Pascal 对于使用可变参数列表的情况是相当严格的。请看下面的例子：

```
Procedure Sam(A:integer=3);overload;
Procedure Sam (A:inreger=3;s:string=' Sam');overload;
```

这个例子造成了不确定的情况，如果使用下面这个调用，那么就搞不清楚会发生什么情况。此时，在编译该程序段时会提示发生“Ambiguous overloaded call to '<name>'”（不明确调用）的错误。



总结

本节除了讲述普通函数的重载问题外，还讲述了某类中的方法的重载以及具有类继承的函数重载问题，包括了如何声明、如何调用以及调用的规则是什么等内容。

第二章

窗体设计

热点透视

窗体在应用程序中占有很重要的地位，它就如同楼房的房屋构架一样，对于窗体的一般应用可以参考其他书籍，本章将主要讲述窗体的一些深层次应用，包括：

- 创建无窗口应用程序
- 显示启动封面
- 实现无标题窗体的移动
- 捕获鼠标在窗体中非客户区的消息
- 防止用户更改窗体位置和大小
- 在任务栏中隐藏应用程序
- 防止关闭 Windows



创建无窗口应用程序



遇到难题

有时程序只是进行一些数据的计算，而无须与用户进行任何交互。此时，程序无须产生一些窗体，也就是说，该程序是一个无窗口的应用程序。



钥匙在此

我们完全可以创建一个没有 Form 的应用程序，此时只需将项目文件中所有的 Form 去掉即可。



跟我来

创建一个新项目，然后使用菜单“FilelRemove Form Project”命令把主 Form 移走。此时的项目文件如下所示：

```
program Project1;

uses
  Forms;

{$R *.RES}

begin
  Application.Initialize;
  Application.Run;
end.
```

事实上，甚至可以把 uses 子句、Application.Initialize 以及 Application.Run 都删掉，此刻的项目文件如下：

```
program Project1;
begin
```

end.

当然，这么简单的程序肯定无任何实际意义，但是，可以在 begin...end 之间加入代码，从而构成 Win32 控制台程序。



显示启动封面



遇到难题

许多 Windows 应用程序在运行时都有一个漂亮的封面，如 Delphi、Word 等。在这些封面上可以告诉用户该程序的版本、版权、使用者以及初始化文件等等信息。其实在我们自己的程序中，也同样可以拥有这样的封面。



钥匙在此

顾名思义，封面是在应用程序启动时显示的，在应用程序初始化期间一直停留在屏幕上。显示封面其实是一件很简单的事情。

设计封面与设计一般的窗体相类似，但创建时又有所不同。首先，要确保该窗体在项目文件中不是自动创建的；其次，要注意显示封面的代码的书写位置。



跟我来

按照下列步骤即可以显示封面：

（1）在创建了主 Form 后，再创建一个 Form 来作为封面。假设这个 Form 叫 SplashForm；

（2）使用“Project|Options”菜单命令，确保 SplashForm 没有出现在 Auto create 列表中；

（3）把 SplashForm 的 BorderStyle 特性设为 bsNone，把 BorderIcons 特性设为[]。

（4）把一个 Timage 元件放到 SplashForm 上，并把它的 Align 特性设为 alClient；

（5）在 Timage 元件中调入一个位图。

设计好封面后，应当在项目文件中加入代码来显示封面。清单 2-1 列出了相应的项目文件，其中包含了显示封面的代码。

清单 2-1　带有启动封面的项目文件

```
program splash;
```

```
uses
  Forms,
  MainFrm in 'MainFrm.pas' {MainForm},
  SplashFrm in 'SplashFrm.pas' {SplashForm};

{$R *.RES}
begin
  Application.Initialize;
  { 创建封面窗体 }
  SplashForm := TSplashForm.Create(Application);
  SplashForm.Show;    // 显示封面
  SplashForm.Update; // 强制更新封面
  {进行延时}
  while SplashForm.tmMainTimer.Enabled do
    Application.ProcessMessages;

  Application.CreateForm(TMainForm, MainForm);
  SplashForm.Hide;   // 隐藏封面
  SplashForm.Free;   // 释放 SplashForm 对象
  Application.Run;
end.
```

上面的代码中有一个循环：

```
  while SplashForm.tmMainTimer.Enabled do
    Application.ProcessMessages;
```

这个循环是用来延时的。SplashForm 上有一个 Ttimer 元件，它的 Interval 属性值为 3000。当 Ttimer 元件的 OnTimer 事件发生时，就执行下面这行代码：

```
  tmMainTimer.Enabled := False;
```

上面这行代码将导致循环结束。



再送礼包

1. 如何通过命令行参数隐藏封面的显示

在一些应用中有时想通过开关参数将封面隐藏起来。这可以通过引入命令行参数来实现。清单 2-2 是清单 2-1 的改进版，它可通过参数将封面隐藏起来，当该参数为“-ns”时将不显示封面。

清单 2-2　由参数来决定是否显示启动封面的项目文件

```
program splash;

uses
   Forms,
   dialogs,Sysutils,
   MainFrm in 'MainFrm.pas' {MainForm},
   SplashFrm in 'SplashFrm.pas' {SplashForm};

{$R *.RES}
begin
   Application.Initialize;
   //根据参数来确定是否显示封面
   if uppercase(paramstr(1))<>uppercase('-ns') then begin
      SplashForm := TSplashForm.Create(Application);
      SplashForm.Show;     // 显示封面
      SplashForm.Update; // 强制更新封面
      {进行延时}
      while SplashForm.tmMainTimer.Enabled do
         Application.ProcessMessages;
    end;

   Application.CreateForm(TMainForm, MainForm);
   if SplashForm<>Nil then    begin
      SplashForm.Hide;   // 隐藏封面
      SplashForm.Free;   // 释放 SplashForm 对象
   end;
   Application.Run;
   end.
```

清单 2-2 中，ParamStr()用来获得该应用程序的参数情况。如果参数返回的是“-ns”时将不显示封面。



在使用 ParamStr 时应注意，该函数的参数是从 0 开始的。ParamStr(0)返回的是应用程序名本身（包括路径），ParamStr(1)表示参数 1，然后依次类推。

2. 如何关闭 Delphi 和 C++ Builder 自身的封面

如果在 Delphi 和 C++ Buider 主程序快捷方式的命令行中加入参数“-ns”(no splash)，那么以后在启动 Delphi 和 C++ Buider 时就不会显示封面了。



实现无标题窗体的移动



遇到难题

在 Windows 应用程序中，要移动一个窗体，一般是通过拖动窗体的标题栏进行的。如果窗体无标题栏，难道就不能拖动该窗体了吗？当然不是，如果要实现无标题栏窗体的拖动，则是需要编写一些代码的。



钥匙在此

为了能够移动无标题窗体，一个最简单的方法是，捕获鼠标发送给应用程序的 WM_NCHITTEST 消息。当鼠标单击窗体的非标题区域时，将该消息发送的结果修改成在单击窗体的标题，以使 Window 以为鼠标是在单击窗体的标题。这样，当拖动该窗体时，Window 将会移动该窗体。



WM_NCHITTEST 消息类型 TWMNCHitTest

类型格式：

```
TWMNCHitTest = packed record
    Msg: Cardinal;
    Unused: Longint;
    case Integer of
      0: (
        XPos: Smallint;
        YPos: Smallint);
      1: (
        Pos: TSmallPoint;
        Result: Longint);
  end;
```

该消息记录中，Msg 为消息常量（或叫消息标识符），Pos 为鼠标的当前坐标位置，Result 为窗口过程返回的值。

Result 代表了当前光标的热点位置，其值为表 2-1 中所列值之一。

表 2-1　　TWMNCHitTest 记录中 Result 的值

值	热点位置说明
HTBORDER	在不可调整大小的窗体边界
HTBOTTOM	在窗体最底端边界
HTBOTTOMLEFT	在窗体的左下角
HTBOTTOMRIGHT	在窗体的右下角
HTCAPTION	在窗体的标题栏
HTCLIENT	在窗体的客户区
HTSYSMENU	在窗体的系统菜单处
HTREDUCE	在窗体的最小化按钮处
其他	

注：该表未全部列出。可参考 Win32 API 中的 WM_NCHITTEST 消息说明。



跟我来

按照下列步骤可以移动无标题窗体：

1）新建一个项目，然后将窗体命名为 FrmMain，并设置 BorderStyle 特性为 bsNone，BorderIcons 特性为[]；

2）在 TfrmMain 接口处声明 WM_NCHITTEST 消息的处理函数（句柄），代码如下：

```
procedure WMNCHitTest(var M: TWMNCHitTest);
        message WM_NCHITTEST;
```

3）为 TFrmMain 定义 WM_NCHITTEST 消息的处理函数，并修改发送该消息的返回结果。

```
procedure TFrmMain.WMNCHitTest(var M: TWMNCHitTest);
begin
  inherited;
  if   M.Result = htClient then
     M.Result := htCaption;
end;
```

完整的程序见清单 2-3。

清单 2-3　移动无标题窗体

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
```

```
type
  TFrmMain = class(TForm)
  private
    { Private declarations }
  public
    procedure WMNCHitTest(var M: TWMNCHitTest);
      message WM_NCHITTEST;
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}
procedure TFrmMain.WMNCHitTest(var M: TWMNCHitTest);
begin
  inherited;
 {如果客户区被鼠标点击时，Window 以为这是在点击标题栏，于是在拖动时，
  Window 认为这是在拖动窗体}
  if   M.Result = htClient then
    M.Result := htCaption;
end;

end.
```



捕获鼠标在窗体中非客户区的消息



遇到难题

我们都知道，鼠标在窗体上移动时会触发窗体的 OnMouseMove 事件，该事件说明如下：

type TMouseMoveEvent = procedure(Sender: TObject; Shift: TShiftState; X, Y: Integer) of object;

property OnMouseMove: TMouseMoveEvent;

从它的声明中不难看出，该事件只是捕获鼠标在窗体区域的坐标位置，也就是说，该事件只对鼠标在窗体的客户区上时才有效，一旦鼠标出了客户区，该事件就无法确定鼠标的位置了。但是，程序员有时需要判断鼠标是否在窗体的标题上或是否在系统菜单上，此时，窗体的 OnMouseMove 事件就帮不上忙了。



钥匙在此

为了获取窗体非客户区（no client）的所有消息，可以参考 Win32 API 帮助中的 WM_NC。本节只是讲述如何捕获鼠标在窗体的非客户区上的移动消息，该消息常量名称为 WM_NCMOUSEMOVE，其消息类型在 Delphi 中被定义为 TWMNCHitMessage。



WM_NCMOUSEMOVE 消息类型 TWMNCHitMessage

类型格式：

```
TWMNCHitMessage = packed record
    Msg: Cardinal;
    HitTest: Longint;
    XCursor: Smallint;
    YCursor: Smallint;
    Result: Longint;
  end;
```

在该消息记录中，Msg 为消息常量，HitTest 为窗口过程处理完消息后返回的鼠标当前位置值，其值可参考表 2-1 中 TWMNCHitTest 记录中的 Result 值。（Xcursor、Ycursor）为鼠标在 Window 桌面上的坐标。Result 无意义。



窗口过程处理完 WM_NCMOUSEMOVE 消息后，返回鼠标的当前位置值，但并没有将它放到 TWMNCHitMessage 的 Result 域中，而是放到了 TWMNCHitMessage 的 HitTest 域中。另外，（Xcursor、Ycursor）为鼠标在 Window 桌面上的坐标，而非相对于窗体的客户区的坐标。



跟我来

按照下列步骤可以捕获鼠标在窗体非客户区的消息：

1）新建一个项目，窗体名为 FrmMain；

2）在 TfrmMain 接口处声明 WM_NCMOUSEMOVE 消息的处理函数（句柄），其代码如下：

```
procedure WMNCMOUSEMOVE(var Message: TWMNCMOUSEMOVE);
        message WM_NCMOUSEMOVE;
```

3）为 TFrmMain 定义 WM_NCHITTEST 消息的处理函数，并修改发送该消息的返回结果。

```
procedure TFrmMain.WMNCMOUSEMOVE(var Message: TWMNCMOUSEMOVE);
begin
…
end;
```

完整的程序见清单 2-4。

清单 2-4　捕获鼠标在窗体非客户区的消息

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TFrmMain = class(TForm)
  private
 procedure WMNCMOUSEMOVE(var Message: TWMNCMOUSEMOVE);
         message WM_NCMOUSEMOVE;
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.WMNCMOUSEMOVE(var Message: TWMNCMOUSEMOVE);
var
  s : string;
begin
  //窗口过程在处理完鼠标在窗体的非客户区中移动的消息后，将返回结果存入
  //消息的 HitTest 域中
```

```
case Message.HitTest of
  HTERROR : s := 'HTERROR';
  HTTRANSPARENT : s := 'HTTRANSPARENT';
  HTNOWHERE : s := 'HTNOWHERE';
  HTCLIENT : s := 'HTCLIENT';
  HTCAPTION : s := 'HTCAPTION';
  HTSYSMENU : s := 'HTSYSMENU';
  HTSIZE : s := 'HTSIZE';
  HTMENU : s := 'HTMENU';
  HTHSCROLL : s := 'HTHSCROLL';
  HTVSCROLL : s := 'HTVSCROLL';
  HTMINBUTTON : s := 'HTMINBUTTON';
  HTMAXBUTTON : s := 'HTMAXBUTTON';
  HTLEFT : s := 'HTLEFT';
  HTRIGHT : s := 'HTRIGHT';
  HTTOP : s := 'HTTOP';
  HTTOPLEFT : s := 'HTTOPLEFT';
  HTTOPRIGHT : s := 'HTTOPRIGHT';
  HTBOTTOM : s := 'HTBOTTOM';
  HTBOTTOMLEFT : s := 'HTBOTTOMLEFT';
  HTBOTTOMRIGHT : s := 'HTBOTTOMRIGHT';
  HTBORDER : s := 'HTBORDER';
  HTOBJECT : s := 'HTOBJECT';
  HTCLOSE : s := 'HTCLOSE';
  HTHELP : s := 'HTHELP';
  else s := '';
end;
//将结果显示在窗体的标题栏中
 FrmMain.Caption := s;
 Message.Result := 0;
end;

end.
```



防止用户更改窗体位置和大小



遇到难题

前面讲述了如何移动无标题窗体，如果反过来情况会是怎么样的呢？即防止移动有标题的窗体，甚至防止用户改变窗体大小，这些同样需要编写代码来实现。



钥匙在此

为了防止用户移动窗体或改变窗体大小，首先应该捕获窗体尺寸变化或者位置变化的消息，然后告诉 Window 是否改变当前窗体的位置或者大小。

当窗体位置或者大小试图发生变化时，该事件就会向该窗体发送 WM_WINDOWPOSCHANGING 消息，当该消息被处理后将会设置消息记录中参数所指向的 WINDOWPOS 结构变量。该结构变量包含有窗体的新位置和大小的信息，同时还可以通过设置该变量的一个标志位来确定系统是否更改窗体的位置或者大小。因此，可以通过捕获 WM_WINDOWPOSCHANGING 消息，然后置标志位来防止更改窗体位置或者大小。WM_WINDOWPOSCHANGING 的消息记录类型为 TWMWINDOWPOSCHANGING，在 Delphi 中与 TWMWindowPosMsg 类型一致。



WM_WINDOWPOSCHANGING 消息类型 TWMWindowPosMsg

类型格式：

```
TWMWindowPosMsg = packed record
  Msg: Cardinal;
  Unused: Integer;
  WindowPos: PWindowPos;
  Result: Longint;
end;
```

该消息记录中，Msg 为消息常量；WindowPos 为指向 WINDOWPOS 结构变量的指针。WINDOWPOS 在 Win32 API 中是这样定义的：

```
typedef struct _WINDOWPOS { // wp
    HWND hwnd;
    HWND hwndInsertAfter;
```

```
    int   x;
    int   y;
    int   cx;
    int   cy;
    UINT flags;
} WINDOWPOS;
```

通过设置该结构变量中的 Flags 域即可实现防止更改窗体大小和位置的功能。Flags 的具体取值可以参考 Win32 API 的 WINDOWPOS 类型变量。其中当 Flags 取值为 SWP_NOMOVE 或 SWP_NOSIZE 时告诉 Window 保留原有的位置和大小。



跟我来

按照下列步骤可以实现防止移动窗体和改变窗体的大小：

1）新建一个项目，窗体名为 FrmMain；

2）在 TfrmMain 接口处声明 WM_WINDOWPOSCHANGING 消息的处理函数（句柄），其代码如下：

```
procedure WMPosChange(var Message: TWMWINDOWPOSCHANGING);
        message WM_WINDOWPOSCHANGING;
```

3）为 TFrmMain 定义 WM_NCHITTEST 消息的处理函数，并修改 WINDOWPOS 的标志位 Flags。

```
procedure TfrmMain.WMPosChange(var Message: TWMWINDOWPOSCHANGING);
begin
  PWindowPos(TMessage(Message).lParam).Flags :=
     PWindowPos(TMessage(Message).lParam).Flags or
     SWP_NOMOVE or
     SWP_NOSIZE;
end;
```

完整的程序见清单 2-5。

清单 2-5　防止改变窗体的位置和大小

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
```

```
  TFrmMain = class(TForm)

  private
    { Private declarations }
    //定义窗体消息 WM_WINDOWPOSCHANGING 的处理函数
    procedure WMPosChange(var Message: TWMWINDOWPOSCHANGING);
        message WM_WINDOWPOSCHANGING;
  public
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TfrmMain.WMPosChange(var Message: TWMWINDOWPOSCHANGING);
begin
  //设置 WINDOWPOS 结构变量的标志位 Flags
  //注：这些标志位可以进行组合赋值
  PWindowPos(TMessage(Message).lParam).Flags :=
      PWindowPos(TMessage(Message).lParam).Flags or
      SWP_NOMOVE or
      SWP_NOSIZE;
end;
end.
```



再送礼包

当然也可用其他的方式来防止改变窗体的大小和位置。如，可以通过设置窗体的 BorderStyle 属性来防止改变窗体的大小和位置，将其设成 bsSingle 时可以防止改变窗体大小，设成 bsNone 时将去掉标题栏并可以防止改变大小和移动位置等。当然这种方法是在损失其他功能的情况下来实现该功能的。



在任务栏中隐藏应用程序



遇到难题

有很多监视程序或者服务程序往往需要暗中运行，而不显示出来，并且还要防止用户关闭该程序。如，某些监视程序为了不让用户知道，或者防止用户通过任务管理器关闭应用程序，需要将自己的程序隐藏起来，从而达到监视用户的使用的目的。如何将程序隐藏起来而不让 Windows 的任务管理器发现呢？



钥匙在此

将应用程序在 Windows 的任务栏中隐藏起来，需要将窗体和应用程序都隐藏起来。为了实现这一目标，只需调用 ShowWindow() 函数隐藏应用程序以及设置 Application.ShowMainForm 的值为 False。



跟我来

现举一个从 Windows 任务栏中隐藏程序的例子，请按照以下步骤进行：

1）选择菜单 “View | Project Source”；

2）在 Uses 语句中加入 Windows 单元；

3）在“Application.Initialize;”之后加入“Application.ShowMainForm := False; ”语句；

4）在“Application.Run;”语句之前加入“ShowWindow(Application.Handle, SW_HIDE);”语句；

其完整的代码类似于以下清单：

```
program Project1;

uses
  Windows,
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};
```

```
{$R *.RES}

begin
   Application.Initialize;
//不显示主窗体
   Application.ShowMainForm := False;
   Application.CreateForm(TForm1, Form1);
   Application.CreateForm(TForm2, Form2);
   ShowWindow(Application.Handle, SW_HIDE);
   Application.Run;
end.
```

在每个窗体单元的 "initialization" 部分加入以下语句：

```
//将应用程序隐藏
   ShowWindow(Application.Handle, SW_HIDE);
```



防止关闭 Windows



遇到难题

对于很多应用程序来说，在 Windows 要进行关闭时，如果当前的作业未存盘或者未处理完时，这些应用程序都会弹出一个是否结束应用程序的对话框，以确定是否真正关闭应用程序。如 Word 程序，在当前编辑的文档未存盘时，如果要关闭 Windows，则 Word 将弹出一个对话框以确定是存储文档，还是取消当前关闭系统的要求；又如 Delphi 中，当正在调试应用程序时，若要关闭 Windows，也会弹出类似的对话框。那么，我们自己的应用程序也可以实现类似的功能吗？



钥匙在此

在其他程序调用了 ExitWindowsEx()时，则会关闭 Windows。如果应用程序正在编辑一个文件而且还未存盘，此时有可能造成数据的丢失，除非应用程序知道 Windows 将要退出。实现该功能其实很简单，只要响应主 Form 的 OnCloseQuery 事件即可。该事件说明如下：

```
type TCloseQueryEvent = procedure(Sender: TObject; var CanClose: Boolean) of object;
```

property OnCloseQuery: TCloseQueryEvent;

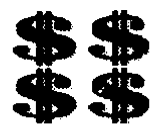
其中，参数 CanClose 决定了该程序是否真正结束本应用程序。当 CanClose 为 True 时，表示将结束本程序，否则，将不结束本应用程序，从而也不会关闭 Windows。



跟我来

请参照下面的代码进行处理：

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
        if MessageDlg('确实要关闭吗？',mtConfirmation,mbYesNoCancel,0) = mrYes then
          CanClose:=true   //表示可以结束本程序
        else
            Canclose:=false;//表示不能结束本程序
end;
```



再送礼包

如果是一个无 Form 的程序，防止结束程序可以采取另一种方式。此时应当捕获 WM_QUERYENDSESSION。只要有一个应用程序调用了 ExitWindowsEx()，每个应用程序就会收到 WM_QUERYENDSESSION 消息。如果应用程序获得这个消息的返回值为非零值，表示允许关闭 Windows。如果返回值为零，表示不允许关闭 Windows。

第三章

MDI 应用

热点透视

"MDI 即多文档界面的缩写，早在 Windows2.0 下的 Microsoft Excel 电子表格程序就开始引入了这个概念，从而，Excel 的用户就可以同时操作几张表格。Windows3.X 的程序管理器以及文件管理器也是 MDI 程序。

本章将讲解 MDI 程序的基本知识，包括如何美化自己的 MDI 主窗体，如何将子窗体较好地隐藏起来，以及如何实现菜单的合并等。读完本章后，您将会有编写 MDI 应用程序的坚实基础。在此基础上，您就可以轻松掌握更高级的技术。本章内容主要包括：

- MDI 应用程序是怎样构成的
- 如何为 MDI 主窗体增加背景
- 如何隐藏子窗体
- 如何合并 MDI 菜单



MDI 应用程序是怎样构成的



遇到难题

在传统的 Windows 编程中，程序员需要知道 Windows MDI Client 类、MDI 数据结构和 MDI 特有的函数和消息。而在 Delphi4 和 Delphi5 中，创建一个 MDI 应用程序就比较简单了。要进行 MDI 应用程序的编程，首先应当掌握 MDI 应用程序是如何构成的。



钥匙在此

要编写 MDI 应用程序，必须先掌握它们是如何构成的。图 3-1 显示了本章将要创建的应用程序主界面。

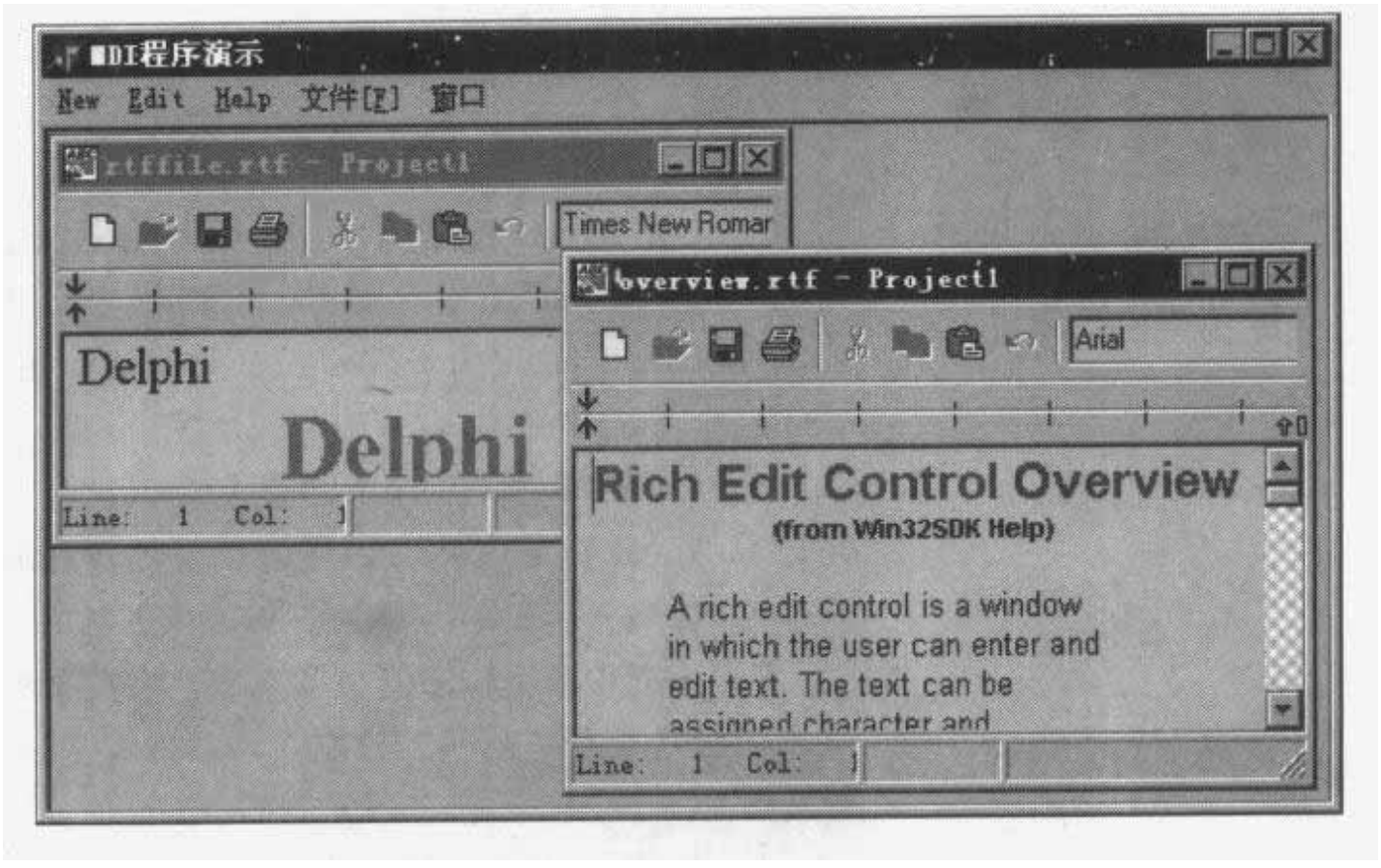


图 3-1　MDI 程序主界面

MDI 应用程序所包含的窗口有：

主窗口　应用程序的主窗口，这个窗口主要用于管理子窗口。该窗口有标题、菜单条和系统菜单。右上角有最小化、最大化和关闭按钮。主窗口的空白区域就是客户区。客户区是在主窗口创建时自动创建的。

子窗口　用于存放数据的窗口，例如文本文件、表格、位图和其他文档等。与主窗口类似，它也有标题栏、系统菜单、最小化、最大化以及关闭按钮等。子窗口的菜单和主窗口

的菜单是有一定联系的。子窗口不能移出客户区。

Delphi4 和 Delphi5 在主窗体管理子窗体的功能上比以往的版本有所增强。例如，为使子窗体进行层叠显示，传统的方法是调用系统函数 SendMessage()来发送 WM_MDICASCADE 消息给客户区。如，

SendMessage(ClientHandle, WM_MDICASCADE, 0 ,0);

而在 Delphi4 和 Delphi5 中使用主窗体的一个方法来实现层叠显示，即只要调用 Cascade() 就行了。

CasCade（）；

下面介绍一个完整的 MDI 应用程序的例子。这个程序主要用于创建或编辑 RTF 文件，就如同 Word 一样，当新建一个 RTF 文档或者打开一个已有 RTF 文档时，将自动创建一个子窗体，同时，主窗体的“窗口”菜单可以对子窗体进行管理。通过该程序，可以很好掌握 MDI 应用程序的主要构架以及如何创建 MDI 应用程序。



跟我来

1. 创建主窗体

新建一个应用程序，然后设置 Form 的名称为 FrmMDIMain，并设置 FormStyle 属性为 fsMDIForm 风格，即表示该窗体为主窗体。

TFrmMDIMain 用于新建窗体、对子窗体进行排列以及关闭程序等。

2. 创建子窗体

添加一个 Form，然后设置 Form 的名称为 MainForm，并设置 FormStyle 属性为 fsMDIChild，即表示该窗体为子窗体。子窗体主要进行文本的编辑和设置字体，并可以使用剪贴板对所选文本进行剪切、复制和粘贴等。

主窗体代码参见清单 3-1。与主窗体有关的子窗体部分源代码参见清单 3-2。

清单 3-1　主窗体代码

```
unit MDIMain;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Buttons, ExtCtrls, Menus;

type
  TFrmMDIMain = class(TForm)
    mmMain: TMainMenu;
    mmiFile: TMenuItem;
    mmiNew: TMenuItem;
```

```
    mmiOpen: TMenuItem;
    N3: TMenuItem;
    mmiExit: TMenuItem;
    mmiWindow: TMenuItem;
    mmiTile: TMenuItem;
    mmiCascade: TMenuItem;
    mmiArrangeIcons: TMenuItem;
    mmiCloseAll: TMenuItem;
    procedure mmiTileClick(Sender: TObject);
    procedure mmiCascadeClick(Sender: TObject);
    procedure mmiArrangeIconsClick(Sender: TObject);
    procedure mmiNewClick(Sender: TObject);
    procedure mmiCloseAllClick(Sender: TObject);
    procedure mmiOpenClick(Sender: TObject);
    procedure mmiExitClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmMDIMain: TFrmMDIMain;

implementation

uses REMain;

{$R *.DFM}

procedure TFrmMDIMain.mmiTileClick(Sender: TObject);
begin
  //并排排列子窗体
  Tile;
end;

procedure TFrmMDIMain.mmiArrangeIconsClick(Sender: TObject);
begin
  //排列最小化的子窗体
   ArrangeIcons;
```

```
end;

procedure TFrmMDIMain.mmiCascadeClick(Sender: TObject);
begin
  //层叠排列
  Cascade;
end;

//新建文档
procedure TFrmMDIMain.mmiNewClick(Sender: TObject);
begin
      with TmainForm.Create(self) do begin
          SetFileName(sUntitled);
          Editor.Lines.Clear;
          Editor.Modified := False;
          SetModified(False);
       end;
end;

    //关闭所有子窗体
procedure TFrmMDIMain.mmiCloseAllClick(Sender: TObject);
var
  i: integer;
begin
  for i := MdiChildCount - 1 downto 0   do
    MDIChildren[i].Close;

end;

//打开文档
procedure TFrmMDIMain.mmiOpenClick(Sender: TObject);
begin
      with TmainForm.Create(self) do begin
          if OpenDialog.Execute then
          begin
                  PerformFileOpen(OpenDialog.FileName);
                  Editor.ReadOnly := ofReadOnly in OpenDialog.Options;
          end;
      end;
end;
```

```
procedure TFrmMDIMain.mmiExitClick(Sender: TObject);
begin
    Close;
end;

end.
```

清单 3-2　子窗体源程序（部分）

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    action:=caFree;
end;

procedure TMainForm.FileNew(Sender: TObject);
begin
        //新建文档
        FrmMDIMain.mmiNewClick(Sender);
end;

procedure TMainForm.FileOpenItemClick(Sender: TObject);
begin
         //打开文档
    FrmMDIMain.mmiOpenClick(Sender);
end;

procedure TMainForm.EditUndo(Sender: TObject);
begin
  //取消修改
    with Editor do
      if HandleAllocated then SendMessage(Handle, EM_UNDO, 0, 0);
  end;

  procedure TMainForm.EditCut(Sender: TObject);
  begin
    //剪切至剪贴板
    Editor.CutToClipboard;
  end;

  procedure TMainForm.EditCopy(Sender: TObject);
```

```
begin
  //拷贝至剪贴板
  Editor.CopyToClipboard;
end;

procedure TMainForm.EditPaste(Sender: TObject);
begin
  //从剪贴板进行粘贴
  Editor.PasteFromClipboard;
end;
```

在主窗体中进行新建或者打开等操作时，都将会创建新的子窗体。因此，为了动态创建子窗体，应设置菜单“project|Option”，以使 MainForm 不自动创建子窗体。

子窗体的一些菜单命令，如“File|New”、“File|Open”等，都调用了主窗体相应的菜单。这主要考虑到菜单合并问题（关于菜单合并问题的内容请参见后面部分）。另外，子窗体中支持对剪贴板的操作以及“Undo”操作（关于剪贴板的操作，请参见“剪贴板”章节）。



在子窗体的 Close 事件中，应当把 Action 参数设为 caFree，以保证程序关闭时自动释放子窗体的实例。这是因为调用 close()时，MDI 子窗体并不是自动关闭的，必须在 OnClose 事件的句柄中决定如何处理这些子窗体。处理该事件需要传递一个变量参数，该参数（Action）可以设为以下四种参数值：

caNone	什么都不做
caHide	隐藏
caFree	释放
caMinimine	最小化（缺省）



为 MDI 主窗体增加背景



遇到难题

在 MDI 程序中，MDI 的主窗口一般的功能是提供子窗口显示的位置和提供菜单、工具条以及状态条等，而窗口的客户区则一般不会有其他的用途，如果在这里画上一些软件的标志、公司的标志或者其他的背景图案的话，不仅可以使 MDI 的主窗口更加充实、美观，而且还可以更加突出公司的形象和增强公司标志在客户心中的地位。对于非 MDI 的 Form 来

说，这是很简单的，只要把一个 Timage 元件放在 Form 上，然后把它的 Alian 属性设为 alClient 就可以了。而对于 MDI 应用程序来说，要在它的主窗体放上一幅图，并不是那么简单。



钥匙在此

如何在主窗体的客户区画图呢？由于 Delphi4 和 Delphi5 的 VCL 没有直接操纵 MDI 客户区的方法，所以只有依靠 Win32 API 了。

实现主窗体客户区的画图主要涉及以下两个环节：

1. 捕获 WM_ERASEBKGND 消息

当应用程序的窗口背景必须被删除时，应用程序将会发送 WM_ERASEBKGND 消息。发送该消息是为了实现在非有效区域内绘图。该消息在 Win32 API 中的定义文法如下：

WM_ERASEBKGND

hdc = (HDC) wParam; // handle of device context

参数：hdc 标识了设备上下文（device context）

返回值：如果删除了背景，则返回值为非 0；否则为 0

2. 绘图

为了将图像信息绘制到主窗体的客户区上，首先需将图像信息复制到客户区中。本节调用的函数为 BitBlt()，该函数可以实现将 Timage 上的图像信息复制到客户区上的功能。BitBlt()在 Win32 API 中的定义文法如下：

```
BOOL BitBlt(

    HDC hdcDest,      // 目标设备上下文句柄
    int nXDest,       // 目标矩形左上角的 X 坐标
    int nYDest,       // 目标矩形左上角的 Y 坐标
    int nWidth,       // 目标矩形的宽度
    int nHeight,      // 目标矩形的高度
    HDC hdcSrc,       // 源设备上下文句柄
    int nXSrc,        // 源矩形左上角的 X 坐标
    int nYSrc,        // 源矩形左上角的 Y 坐标
    DWORD dwRop       // 光栅操作代码
  );
```

其中，dwRop 决定了源设备与目标设备的图形颜色数据如何进行组合，从而实现目标设备的最终图形颜色数据。如：

SRCCOPY　表示直接将源数据拷贝至目标数据中

SRCAND　表示源数据与目标数据进行“与”运算，并将结果存成目标数据

其余操作代码参见 Win32 API。



跟我来

由于 MDI 主窗口的特性，使用普通 OnPaint 或者使用 TImage 等方法都不会产生作用。下面将编写一个简单的 MDI 程序来介绍如何实现在主窗体的客户区画图的功能。

第一步：运行 Delphi，打开程序 3-1。

第二步：在 FrmMDIMain 上增加一个 Image 元件，选择要设置的背景并放到 Image 的 Picture 属性中。

第三步：在 FrmMDIMain 的 Private 中定义：

```
FOldClientProc,
FNewClientProc: TFarProc;
FDrawDC: hDC;
procedure DrawTiled;
procedure ClientWndProc(var Message: TMessage);
procedure CreateWnd; override;
```

第四步：在实现（implementation）中加入上述过程的具体内容：

```
//当 Form 创建时调用
procedure TFrmMDIMain.CreateWnd;
begin
  inherited CreateWnd;
  // 使 ClientWndProc 方法成为一个有效的窗口过程
  FNewClientProc := MakeObjectInstance(ClientWndProc);
  // 获取原窗口过程指针
  FOldClientProc := Pointer(GetWindowLong(ClientHandle, GWL_WNDPROC));
  // 将 ClientWndProc 设成新窗口过程
  SetWindowLong(ClientHandle, GWL_WNDPROC, LongInt(FNewClientProc));
end;

procedure TFrmMDIMain.ClientWndProc(VAR Message: TMessage);
begin
  case Message.Msg of
    // 捕获 WM_ERASEBKGND 消息，并实现客户区的绘图
    WM_ERASEBKGND:
      begin
        CallWindowProc(FOldClientProc, ClientHandle, Message.Msg, Message.wParam,
          Message.lParam);
        //获取主窗口的客户区句柄
        FDrawDC := TWMEraseBkGnd(Message).DC;
```

```
        DrawTiled;
        Message.Result := 1;
    end;
  //捕获滚屏消息，确保客户区的刷新
  WM_VSCROLL, WM_HSCROLL:
    begin
        Message.Result := CallWindowProc(FOldClientProc, ClientHandle, Message.Msg,
        Message.wParam, Message.lParam);
        InvalidateRect(ClientHandle, Nil, True);
    end;
  else
  //缺省地调用原先的窗口过程
        Message.Result := CallWindowProc(FOldClientProc, ClientHandle, Message.Msg,
          Message.wParam, Message.lParam);
    end;
end;

procedure TFrmMDIMain.DrawTiled;
{ 将图像信息平铺于主窗体的客户区内}
var
    Row, Col: Integer;
    CR, IR: TRect;
    NumRows, NumCols: Integer;
begin
    GetWindowRect(ClientHandle, CR);
    IR := imgMain.ClientRect;
    NumRows := CR.Bottom div IR.Bottom;
    NumCols := CR.Right div IR.Right;
    with imgMain do
      for Row := 0 to NumRows+1 do
        for Col := 0 to NumCols+1    do
            //将 imgMain 中的图像信息复制到主窗体的客户区中
            BitBlt(FDrawDC, Col * Picture.Width, Row * Picture.Height,
                    Picture.Width, Picture.Height, Picture.Bitmap.Canvas.Handle,
                    0, 0, SRCCOPY);
end;
```

编译运行这个程序后，将会发现，Image 元件并不会在 Form 上显示出来，但是整个 Form 的客户区域被 Image 中的图像所铺满。运行结果如图 3-2 所示。



图 3-2　具有背景的 MDI 应用程序



隐藏子窗体



遇到难题

在 Delphi 中，如果试图使用如下语句来隐藏 MDI 程序的子窗体，则会返回错误信息。

```
ChildForm.Hide;
```

这是因为，以这种方式来隐藏子窗体是不允许的。Windows 在 MDI 的实现上有缺陷，因此隐藏子窗体会破坏子窗体的层叠次序（z-order）。除非在使用这个技术时非常小心，否则，试图隐藏子窗体会给应用程序带来混乱。然而，有时候又确实需要隐藏子窗体，怎么办？



钥匙在此

有两种方法可以实现隐藏子窗体，但使用起来千万要小心。

一种方法是不画出子窗体。可以调用 LockWindowUpdate()来禁止画出 MDI 子窗体。当想要创建一个子窗体而又不想显示出来时，禁止画出子窗体就很有用。例如，假设一个子窗体是用来查询数据库的，如果查询失败就可能要释放该窗体。这时除非使用了一些技术来隐藏窗体，否则，将看到窗体在屏幕上闪烁。LockWindowUpdate()可以用来指定窗口的重画。

另一种方法，就是调用 Win32 API 函数 ShowWindow()。通过指定 SW_HIDE 参数就可以隐藏窗体。以后要恢复这个子窗体，必须调用 SetWindowPos()函数。这个技术适用于已经创建或者已经显示给用户的窗体。

下面介绍这三个函数的使用方法：

1）LockWindowUpdate()

该函数使重画在指定的窗体中无效或重新有效。它一次只能锁定一个窗体。在 Win32 API 中的定义文法如下：

```
BOOL LockWindowUpdate(

    HWND hWndLock // 锁定窗口的句柄
  );
```

如果函数运行成功，则返回值为非 0。当一个带锁定窗口的应用程序调用 GetDC、GetDCEx 或 BeginPain 函数时，将返回一个带有空的但又可见的区域的设备上下文。这种状态持续到应用程序调用 LockWindowUpdate()对窗体进行解锁时为止，一个锁定的窗体不能被移动。

2）ShowWindow()

该函数用于设定指定窗口的显示状态。在 Win32 API 中的定义文法如下：

```
BOOL ShowWindow(
    HWND hWnd,// 窗体句柄
    int nCmdShow      // 设置窗体的显示状态
  );
```

nCmdShow 可以取很多常量，具体值可参见 MS SDK 文档。

3）SetWindowPos()

该函数用于改变子窗体、弹出式窗体或顶层窗体的大小、位置和次序（Z order）等。这些窗体是根据它们在屏幕上出现的先后来排序的。最顶端的窗体是 Z order 顺序中的第一个窗体并将获得最高等级。在 Win32 API 中的定义文法如下：

```
BOOL SetWindowPos(
    HWND hWnd,// 窗口句柄
    HWND hWndInsertAfter,     // 放置顺序句柄
    int X,     // 水平位置
    int Y,     // 垂直位置
    int cx,    // 宽度
    int cy,    // 高度
    UINT uFlags  // 窗体定位标志
  );
```

其中，参数 hWndInsertAfter 标识了 Z order 中在定位窗体之前的窗体。这个参数必须是一个窗体句柄或者是取以下值之一：

HWND_BOTTOM 将窗体放置在 Z order 的最底端。如果 hWnd 参数标识了最顶端窗体，则该窗体将失去最顶端的状态而被放置在所有其他窗体之后。

HWND_NOTOPMOST 将窗体放在所有非顶层窗体之上，同时在所有顶层窗体之后。如果窗体已经是一个非顶层窗体，则该标志无效。

HWND_TOP 将窗体放置在 Z order 之首。

HWND_TOPMOST 将窗体放在所有非顶层窗体之上。该窗体一直维持其顶层位置。

参数 uFlags 指定窗体大小和位置。这个参数可以是多个值的组合。具体参数值可参见相关文档。如 SWP_NOSIZE 参数表示保持窗体现有大小。



跟我来

清单 3-3 显示了上述两种隐藏子窗体的技术。该程序中创建了两种窗体表单，其中主窗体表单类名为 TmainForm，子窗体表单类名为 TMDIChildForm。

清单 3-3　隐藏子窗体

```
unit MainFrm;

interface

uses Windows, SysUtils, Classes, Graphics, Forms, Controls, Menus,
   StdCtrls, Dialogs, Buttons, Messages, ExtCtrls, ComCtrls, MdiChildFrm;

type
   TMainForm = class(TForm)
      mmMain: TMainMenu;
      mmiFile: TMenuItem;
      mmiNew: TMenuItem;
      mmiClose: TMenuItem;
      mmiWindow: TMenuItem;
      N1: TMenuItem;
      mmiExit: TMenuItem;
      mmiHide: TMenuItem;
      mmiShow: TMenuItem;
      mmiHideForm: TMenuItem;
      procedure mmiNewClick(Sender: TObject);
      procedure mmiCloseClick(Sender: TObject);
      procedure mmiExitClick(Sender: TObject);
      procedure mmiHideClick(Sender: TObject);
      procedure mmiShowClick(Sender: TObject);
      procedure mmiHideFormClick(Sender: TObject);
private
      procedure CreateMDIChild(const Name: string);
public
      HideForm: TMDIChildForm;       //用于存放隐藏的子窗体
      Hidden: Boolean;
end;
```

```
var
  MainForm: TMainForm;

implementation

{$R *.DFM}

procedure TMainForm.CreateMDIChild(const Name: string);
var
  MdiChild: TMDIChildForm;
begin
  MdiChild := TMDIChildForm.Create(Application);
  MdiChild.Caption := Name;
end;

procedure TMainForm.mmiNewClick(Sender: TObject);
begin
  CreateMDIChild('非隐藏窗体' + IntToStr(MDIChildCount + 1));
end;

procedure TMainForm.mmiCloseClick(Sender: TObject);
begin
  if ActiveMDIChild <> Nil then
    ActiveMDIChild.Close;
end;

procedure TMainForm.mmiExitClick(Sender: TObject);
begin
  Close;
end;

procedure TMainForm.mmiHideClick(Sender: TObject);
begin
  if Assigned(HideForm) then //测试 HideForm 是否已被赋值
    ShowWindow(HideForm.Handle, SW_HIDE);//将该窗体隐藏
  Hidden := True;
end;

procedure TMainForm.mmiShowClick(Sender: TObject);
begin
```

```
    if Assigned(HideForm) then
      SetWindowPos(HideForm.handle, HWND_TOP, 0, 0, 0, 0, SWP_NOSIZE
        or SWP_NOMOVE or SWP_SHOWWINDOW); //将隐藏窗体显示出来
    Hidden := False;
  end;

  procedure TMainForm.mmiHideFormClick(Sender: TObject);
  begin
    if not Assigned(HideForm) then
    begin
      if MessageDlg('是否创建隐藏窗体?', mtConfirmation, [mbYes, mbNo], 0) = mrYes then
      begin
       //锁定主窗体，使其不画出子窗体
       LockWindowUpdate(Handle);
     Try
       //创建子窗体并赋值给 HideForm
       HideForm := TMDIChildForm.Create(Application);
       HideForm.Caption := '隐藏窗体';
       ShowMessage('隐藏窗体已经创建. 按“OK”按钮将显示窗体');
     finally
       //解锁，使子窗体显示有效
       LockWindowUpdate(0);
     end;
  end
  else begin
    HideForm := TMDIChildForm.Create(Application);
    HideForm.Caption := '隐藏窗体';
  end;
    end
    else if not Hidden then
        HideForm.SetFocus;
  end;

  end.
```

在该程序创建隐藏窗体 mmiHideFormClick()过程中，创建的子窗体既可以直接隐藏和显示，也可以在显示之后由用户隐藏。对于第一种情况，是通过锁定主窗体来禁止子窗体的画出，再通过解锁来显示“隐藏”的子窗体；对于第二种情况，先创建一个子窗体，然后由用户的菜单命令 mmiHideClick()和 mmiShowClick()分别来隐藏和显示子窗体，其中隐藏窗体使用了 ShowWindow()函数，显示窗体使用了 SetWindowPos()函数。作者建议使用第二种

技术。



合并 MDI 菜单



遇到难题

在使用 MDI 应用程序时，常常会看到，当新建了一个子窗体后主窗体的菜单发生了变化，一般情况下菜单是增加了，此时的菜单一部分是原先主窗体自身所拥有的，另一部分是子窗体所拥有的，而主窗体的菜单不会被子窗体菜单所覆盖，这就是“菜单合并”。在 MDI 应用程序中，使用菜单并不比在其他程序中使用菜单困难，但还是稍有不同。下面将介绍如何通过“菜单合并”技术使子窗体能够共享主窗体的菜单栏。



钥匙在此

TMenuItem.GroupIndex 属性控制着主窗体与其他窗体菜单的合并。

TMenuItem.GroupIndex 属性标识了菜单项属于哪一个逻辑组。使用该属性可以控制菜单是如何合并的。在拥有多个窗体的应用程序中，第二个窗体的菜单项在这个窗体被激活时可以被并入相应用程序的主菜单中。MDI 应用程序总是将子窗体的菜单嵌入到主窗体的菜单中。

所有在菜单栏中的菜单项（主菜单）都有一个同样的缺省 GroupIndex 值。使用该属性合并菜单，只需设置每一个连续的菜单项的 GroupIndex 值等于或大于前一个菜单项的 GroupIndex 值即可。

当被合并到应用程序主菜单的菜单项与主菜单中原有的菜单项有相同的值时，主菜单中的菜单项将被相应的菜单项所替代。如果在应用程序主菜单中有多个相同 GroupIndex 值的菜单项，它们都将被其他窗体中具有相同 GroupIndex 值的菜单项所替代。因此，第一个带有给定值的主菜单项将被子窗体拥有同样值的第一个菜单项所替代，第二个带有给定值的主菜单项将被子窗体拥有同样值的第二个菜单项所替代，依次类推。

当被合并到应用程序主菜单的菜单项的 GroupIndex 值介于应用程序主菜单两个临近菜单项的 GroupIndex 值时，该菜单项将被插入到这两个菜单项之间。如果大于所有主菜单菜单项的 GroupIndex 值时，该菜单项将插入到主菜单的最后。

因此，对于 MDI 应用程序来说，菜单的合并是自动的，只要正确地设置了菜单项的 GroupIndex 属性，则当一个子窗体被激活时，菜单就能正确地合并。对于非 MDI 程序来说，当将其他窗体的菜单 TMainMenu.AutoMerge 属性设为 True 时，也可以实现菜单的自动合并。



跟我来

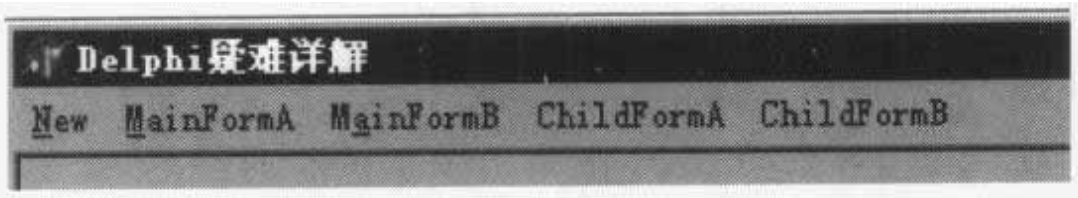
创建一个主窗体，首先在其上放置一个菜单元件，添加菜单项并设置所有菜单项的 GroupIndex 值为 0；其次，创建一个子窗体，同样也放置一个菜单元件，添加菜单项并设置所有菜单项的 GroupIndex 值为 1。图 3-3 显示了主窗体菜单、子窗体菜单以及运行后创建子窗体时的菜单。



主窗体菜单



子窗体菜单



合并后的菜单项

图 3-3　菜单合并

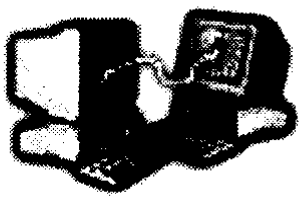
第四章

菜单

热点透视

菜单是 Windows 应用程序中使用得最普遍的标准控件之一。在 Delphi 中已将菜单封装成标准控件供程序员使用，因此简化了编程的复杂度。但是，也正因为封装的原因，使得菜单的一部分底层功能无法使用，所以只有借助于系统消息或者 Win32 API 来实现这些功能。如，添加系统菜单项、截获菜单的非加速键等。另外，随着 Delphi 本身功能的不断加强，使得程序中使用菜单的编程更加容易和合理化，如菜单项与过程的动态关联等等。针对这些问题，本章将讲述：

- 添加系统菜单项
- 为菜单项增加提示功能
- 截获菜单的非加速键
- 动态创建菜单项
- 在菜单中动态关联过程



添加系统菜单项



遇到难题

在应用程序中，系统菜单中包括了对窗体的一般操作，如恢复、移动、最大化、最小化、关闭等，这些操作对于每一个 Windows 应用程序来说都是缺省含有的。然而有时确实有必要在系统菜单中增加自己的菜单命令，那么如何来实现呢？



钥匙在此

在系统菜单中增加自己的菜单项，应该考虑两个问题，一是如何在系统菜单中增加菜单项；二是如何对自定义的系统菜单项进行响应，即当用户单击自定义菜单项时，应捕获该事件。

1. AppendMenu()函数

可以调用 Win32 API 中的 AppendMenu()函数来在系统菜单中增加菜单项。该函数的定义文法如下：

```
BOOL AppendMenu(

    HMENU hMenu,    // 要改变的菜单栏的句柄
    UINT uFlags,  // 菜单项标志位
    UINT uIDNewItem,      // 菜单项标识符或下拉菜单或子菜单句柄
    LPCTSTR lpNewItem    // 菜单项内容
);
```

其中，参数 uFlags 是控制新菜单项外观和行为的标志。这个参数可以是多个常量值的组合。例如，MF_STRING 标志指定菜单项为文本字符串，MF_CHECKED 将放置一个“√”于菜单项之前，如果该参数设成 MF_STRING +MF_CHECKED，则表示在文本菜单项前增加一个“√”符号。

参数 uIDNewItem 指定菜单项标识符，或在 uFlags 为 MF_POPUP 时指定下拉菜单或子菜单句柄。参数 lpNewItem 指定新菜单项的内容。

对于菜单的其他操作可以通过调用 GetSystemMenu、AppendMenu、InsertMenu 以及 ModifyMenu 等 Win32 API 函数实现。

2. WM_SYSCOMMAND 消息

为了解决第二个问题，应用程序应当捕获 WM_SYSCOMMAND 消息。当用户单击自定义菜单项时，程序应当对 WM_SYSCOMMAND 消息进行处理。

当用户从窗体菜单（可以是系统菜单或控制菜单）中选择一个命令或者选择最大化或最小化按钮时，窗体可以收到 WM_SYSCOMMAND 消息，消息定义文法如下：

```
WM_SYSCOMMAND
uCmdType = wParam;          // 系统命令请求类型
xPos = LOWORD(IParam);      // 在屏幕坐标中的水平位置
yPos = HIWORD(IParam);      // 在屏幕坐标中的垂直位置
```

参数 uCmdType 为系统命令请求类型，可以取值为 SC_CLOSE（关闭窗体）和 SC_SIZE（更改窗体大小）等等类型，具体可以参见 MS SDK 说明。xPos 和 yPos 为鼠标的位置。



WM_SYSCOMMAND 消息中，uCmdType 参数的前四位已被 Window 内部使用了。因此，为了获得正确的结果，应用程序应确保 uCmdType 大于 0xFFF0，这可以通过 uCmdType 与 0xFFF0 进行位的“与”操作来实现。



跟我来

新建一个应用程序。清单 4-1 为增加系统菜单项的完整代码。

清单 4-1　增加系统菜单项

```
unit Main;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus;

type
 TFrmMain = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    procedure WMSysCommand(var Msg: TWMSysCommand);
      message WM_SYSCOMMAND;
  public
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;
```

```
implementation

{$R *.DFM}

const
  SC_MyMenuItem = WM_USER + 1;

procedure TFrmMain.FormCreate(Sender: TObject);
begin
  //在系统菜单中添加一个分割符
  AppendMenu(GetSystemMenu(Handle, FALSE), MF_SEPARATOR, 0, '');
  //在系统菜单中添加自己的菜单项
  AppendMenu(GetSystemMenu(Handle, FALSE),
             MF_STRING+MF_CHECKED,
             SC_MyMenuItem,
             '我的菜单项[&I]');
end;

procedure TFrmMain.WMSysCommand(var Msg: TWMSysCommand);
begin
  if Msg.CmdType = SC_MyMenuItem then
    ShowMessage('获得消息')
  else
    inherited;
end;

end.
```

运行该程序代码，然后单击系统菜单，此时将弹出如图 4-1 所示的系统菜单界面。当单击“我的菜单项时”，应用程序将弹出“获得消息”的对话框，表示应用程序已响应该消息。

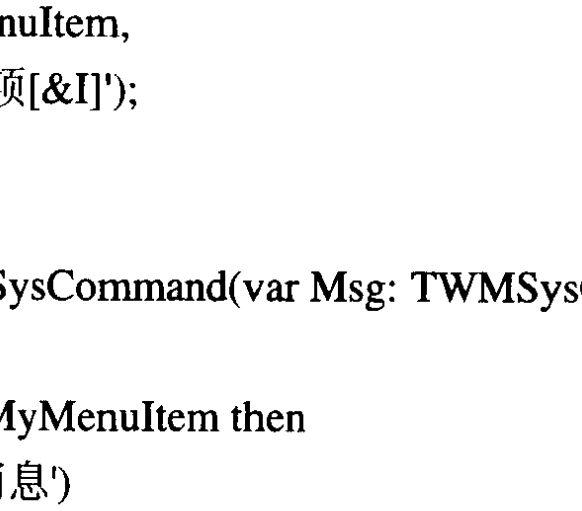


图 4-1　增加系统菜单



为菜单项增加提示功能



遇到难题

当鼠标在菜单上移动时，程序的状态条上会出现该菜单项的功能介绍，即该菜单项是用来做什么的。因为菜单项上的名称往往比较简练，不能很全面的介绍该菜单项的功能，因此需要通过状态条将该菜单功能加以描述。这是一种较好的编程风格，我们也可以将此功能加入到自己编写的应用程序中。



钥匙在此

显示菜单项的功能提示在 Delphi 中实现起来很简单，可以说只需短短的几行代码。对于 Delphi 应用程序来说，当鼠标指针越过控件或菜单项时，会触发应用程序的 OnHint 事件 TApplication.OnHint。由于 TApplication 为非可视对象，并且 Delphi 已将该事件封装好了，因此无法在设计状态条时通过对象游览器的事件来编写代码，但可以通过在应用程序运行时将显示提示的函数句柄赋值给 Application.OnHint 来实现，即

Application.OnHint :=myDisplayHintEventHandler。



跟我来

清单 4-2 实现了将菜单项的提示显示在状态条中。

清单 4-2　为菜单项增加提示功能

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus, ExtCtrls, ComCtrls, StdCtrls;

type
```

```
TFrmMain = class(TForm)
    MainMenu1: TMainMenu;
    MenuItemFile: TMenuItem;
    MenuItemOpen: TMenuItem;
    MenuItemClose: TMenuItem;
    OpenDialog1: TOpenDialog;
    StatusBar1: TStatusBar;
    procedure FormCreate(Sender: TObject);
    procedure MenuItemCloseClick(Sender: TObject);
    procedure MenuItemOpenClick(Sender: TObject);
  private
      { Private declarations }
     procedure HintHandler(Sender: TObject);
    public
      { Public declarations }
    end;

var
    FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.FormCreate(Sender: TObject);
begin
    MenuItemFile.Hint := 'File 菜单';
    MenuItemOpen.Hint := '打开一个文件';
    MenuItemClose.Hint := '关闭应用程序';
    //将自定义提示函数句柄赋值给 Application.OnHint 属性
    Application.OnHint := HintHandler;
end;

//自定义提示函数
procedure TFrmMain.HintHandler(Sender: TObject);
begin
    StatusBar1.Panels[0].Text := Application.Hint;
end;
```

```
procedure TFrmMain.MenuItemCloseClick(Sender: TObject);
begin
   Application.Terminate;
end;

procedure TFrmMain.MenuItemOpenClick(Sender: TObject);
begin
   if OpenDialog1.Execute then
      Caption := OpenDialog1.FileName;
end;
```

end.

如果在窗体创建时就将自定义提示函数句柄赋值给了 Application.OnHint，那么在鼠标指针越过菜单项时就会运行自定义的提示函数。运行该程序后，当鼠标指针移动到菜单项上时，会出现如图 4-2 所示的结果。

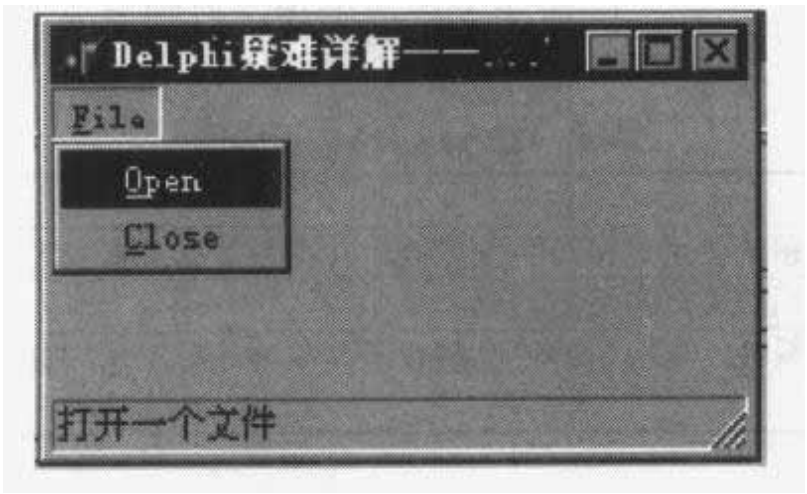


图 4-2　为菜单项增加提示功能



截获菜单的非加速键



遇到难题

在使用菜单时，为了使菜单操作更为方便，除了使用快捷键之外，还可以使用加速键，即可以通过按【Alt】键+加速字符（菜单项中带有下划线的字符）来打开相应的主菜单，如通过按【Alt+F】即可打开“File”主菜单；或者可以在菜单打开时按加速键执行相应的菜单命令。如果按的字符键为非菜单项加速字符时，应用程序一般都不做任何反应。有时，在按

了非加速键时需要应用程序对此给以提示或警告，如何实现这一功能呢？



钥匙在此

应用程序在运行时，当按了【Alt+非加速键】，或者在菜单打开时按了非加速键，如果此时需要给以提示，则只要在应用程序中截获相应的消息即可实现。该消息为WM_MENUCHAR，当菜单为激活状态（按了【Alt】键或者主菜单项打开时为激活状态），而此时如果用户按了非加速键，系统将向应用程序发送 WM_MENUCHAR 消息，其定义的文法如下：

```
WM_MENUCHAR
chUser = (char) LOWORD(wParam); // ASCII 字符
fuFlag = (UINT) HIWORD(wParam); // 菜单标志位
hmenu = (HMENU) lParam;          // 菜单句柄
```

其中，参数 chUser 为 wParam 的低位值（low-order word），表示用户所按下的 ASCII 字符；参数 fuFlag 为 wParam 的高位值（high-order word），表示被激活菜单的类型，可以是 MF_POPUP 或 MF_SYSMENU，前者为下拉菜单、子菜单或者快捷菜单，后者为系统菜单。应用程序在处理完该消息后，在返回值的高位值中应该返回表 4-1 中所示的值之一：

表 4-1　　WM_MENUCHAR 返回值

值	意义
0	通知系统使用户按的字符无效，并在系统扬声器中产生一个短暂的嘟嘟声
1	通知系统关闭激活的菜单
2	通知系统，返回值的低位值表示菜单项相对于 0 的位置。这个菜单项被 Windows 所选择

下面看一下该消息的实际应用的例子。



跟我来

清单 4-3 是截获菜单非加速键程序的主体部分。

清单 4-3　截获菜单的非加速键

```
type
  TFrmMain = class(TForm)
    MainMenu1: TMainMenu;
    One1: TMenuItem;
    One2: TMenuItem;
    One3: TMenuItem;
    One11: TMenuItem;
    One12: TMenuItem;
```

```
    One21: TMenuItem;
    One22: TMenuItem;
  private
    procedure WmMenuChar(var m : TMessage); message WM_MENUCHAR;
  public
  end;

...
procedure TFrmMain.WmMenuChar(var m : TMessage);
begin
  Caption := '您按了'+char(m.wParam)+'键';
  m.Result := 2;
end;
```

运行该程序后，如果按了【Alt+非菜单加速键】，或在打开了主菜单时按了非加速键，则窗体的标题栏中将显示出所按的字符键。



动态创建菜单项



遇到难题

TmenuItem 部件拥有许多属性，通过设置这些属性可以很容易地在程序运行期间改变菜单。您可能希望根据情况呈现不同的菜单，这对编写那些在给定的条件下使某些菜单项无效的程序是很有用的。比如，如果在应用程序中没有文件“打开”，那么文件“存储”和“另存为”菜单项应该无效。另一个常见的动态菜单的例子是“编辑”菜单上普遍具有的“剪切”、“拷贝”和“粘贴”菜单项，“剪切”和“拷贝”通常是无效的，除非有什么对象被选中，而“粘贴”除了有东西放在了剪贴板上外都是无效的。下面是以这种方法来设置“编辑”菜单的一个过程（从该菜单项的 OnClick 事件处理程序中调用该过程）：

```
procedure TEditForm.UpdateMenus;
var
    HasSelection:Boolean;
begin
    paste1.Enabled:=ClipBoard.hasFormat(CF_TEXT);
    HasSelection:=Memo1.SelLength<>0;
```

```
    Cut1.Enabled:=HasSelection;
    Copy1.Enabled:=HasSelection;
    Delete1.Enabled:=hasSelection;
end;
```

您可能还希望设计这样一个程序，在该程序中能根据用户的“许可”级别，限定某些特定的用户不能看见某些菜单。应用程序可以识别“普通用户”和“系统管理员”，而后者要有一个专用口令才能注册。只有“系统管理员”才知道有“权限管理”菜单。这需要动态菜单技术来显示用户所能拥有的菜单。



钥匙在此

利用动态菜单技术来提高程序的适用性是很好的，但满屏菜单都弄成经常出现和消失却不是一个好主意。当与菜单相关的功能不适用时，使相应的菜单项无效要比把它隐藏好。当其无效时，用户仍可以看见该菜单，并且知道该程序支持这种功能。不过，不应该一概都使用这种方法，譬如这里就不应该让用户知道该程序还有“权限管理”功能。

下列 TmenuItem 属性对于动态菜单的处理是很有用处的。

Break	允许给菜单增加一个分隔符，其可能的取值为： MbNone（缺省值） MbBarBreak（把菜单项放到一个新的具有一行分隔符的列中） MbBreak（把菜单项放到一个新的没有分隔符的列中）
Caption	显示在菜单上的文本
Checked	表示有无可选标记出现在该菜单项旁边
Command	可以用于为在运行期间通过截获 Windows/Form 的消息流而产生的菜单项激活事件代码
Count	表示由 TmenuItem 或 TmainMenu 包含的菜单项的数目，只用于程序运行期间
Enabled	表示一个用户可否访问该菜单项的布尔值，无效的菜单项是灰色的
Items	通过子菜单项的索引号访问该子菜单，当应用于一个 TmaniMenu 时，可以通过索引号访问菜单栏的项。只用于程序运行期间。
ShortCut	对分配给 TmenuItem 的快捷键提供读/写访问

Visible　设置一个菜单或主菜单对用户是否可见的布尔值。注意，不可见的菜单并不是无效的——如果给一个不可见的子菜单的菜单项定义了快捷键，那么此时该快捷键仍然起作用。这是在没有任何相关菜单显示的程序中定义全程热键的便捷方法



对 Command 的补充

Command、Count 和 Items 都是程序运行期间使用的属性，因此无法在设计时对其进行设置。并且 Command 是只读的——VCL 给菜单项分配了大小不一的专属命令 ID，以保证程序中的每个菜单项都有一个唯一的 ID。一个菜单项的命令 ID，可以在程序的一次运行到下一次运行，或者从一个表单的一次执行到下一次执行时，做出相应的改变但同时保持它的唯一性。

与 TmenuItems 相关联的一个事件处理程序是 OnClick。下面是关于如何在运行期间为所创建的菜单产生事件代码的介绍（由于它们不能访问 OnClick 事件处理程序）。

下列方法用于动态的改变菜单显示：

Add　在一个现有的菜单末尾增添一个菜单项

Insert　将一个菜单项插入到指定的位置

Remove　删除一个菜单项



跟我来

程序清单 4-4 是一个在 mnuManual 主菜单项上动态创建 TmenuItems 的程序。

清单 4-4　动态创建菜单

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Menus;

type
  TFrmMain = class(TForm)
    MainMenu1: TMainMenu;
    mnuManual: TMenuItem;
    PopupMenu1: TPopupMenu;
    procedure FormShow(Sender: TObject);
  private
```

```
    { Private declarations }
  public
      procedure myClick(Sender: TObject);
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.FormShow(Sender: TObject);
var
   scrfiles,I: Integer;
   SearchRec: TSearchRec;
   OurItem:TMenuItem;
begin

     {为每一个屏幕保护程序创建 OurItem }
     scrfiles := FindFirst ('C:\Windows\system\*.scr', faAnyFile,SearchRec);
try
     while scrfiles = 0 do          {当没有更多的屏幕保护程序文件时停止查找}
     begin
          OurItem :=TMenuItem.Create(Self);
          OurItem.Caption := SearchRec.Name;
          OurItem.OnClick:=myClick;
          mnuManual.Add(OurItem);
          scrfiles := FindNext(SearchRec);
       end;
  finally
       FindClose(SearchRec);
  end;
  end;

  procedure TFrmMain.myClick(Sender: TObject);
  begin
     showmessage('屏幕保护程序文件名:'+TMenuItem(sender).caption);
  end;
```

end.

该过程表示在用户的 C:\Windows\System 目录中查找屏幕保护程序文件（*.scr）。对于该目录中的每一个.scr 文件，它都能创建一个以该文件名为标题的 TmenuItem，并且把它增加到 mnuManual 的底部，如图 4-3 所示。当单击相应的菜单项时将弹出对话框显示“屏幕保护程序文件名:”+文件名。可以使用 Insert 而不是 Add 去改变这个新菜单的位置。这种类型的过程在往一个程序中增加不同的可选模块时大有用处。创建动态菜单可以使用户在一个应用程序中建立他们喜欢的菜单。

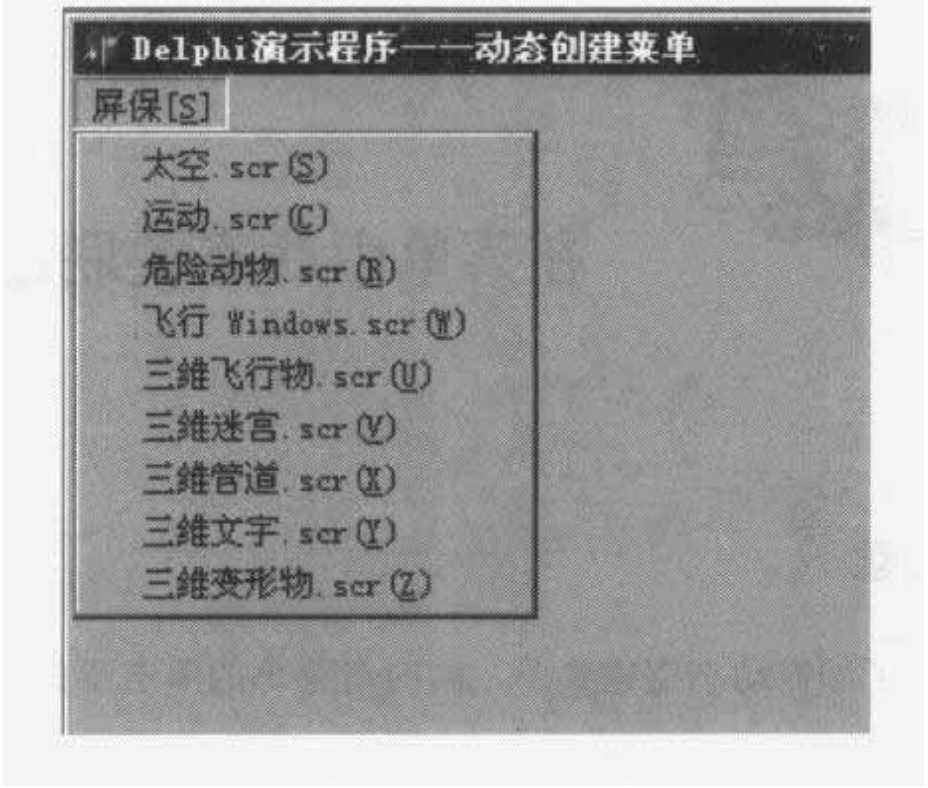


图 4-3　动态创建菜单



再送礼包

文件查找函数组：FindFirst()、FindNext()、FindClose()

这些函数组都创建于 SysUtils 单元中。FindFirst（）用于在指定的目录中查找第一个与给定的特征串（目录和文件名标记，可以包括通配符和一个属性码）相匹配的文件。产生的所有文件信息都放在一个 TsearchRec 类型的变量段中。FindNext（）必须使用 FindFirst（）中使用的 TsearchRec 变量。如果 FindNext 函数成功地找到匹配文件，则其返回值为 0，否则便返回一个 Windows 错误代码。最后调用 FindClose（）释放被 FindFirst()所分配的内存。

FindFirst()

1）函数定义格式：

function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer;

该函数的查找路径由路径参数 Path 和属性参数 Attr 所指定。查找结果放入变量 F 中。如果找到所匹配的文件，函数返回值为 0，否则，将返回一个 Windows 错误代码。其中，参数 Path 可以使用通配符，如“c:\test*.*”表示查找 C:\test 目录下的所有文件。

2）FindNext()

函数定义格式：

function FindNext(var F: TSearchRec): Integer;

该函数将返回与 FindFirst()中所指定的文件相匹配的下一个文件入口，并将该结果放入变量 F 中，注意该变量必须是 FindFirst()中所使用的查找记录。如果成功则返回 0，否则返回一个 Windows 错误代码。例如，在清单 4-4 中，当程序运行了代码

scrfiles := FindFirst ('C:\Windows\system*.scr', faAnyFile,SearchRec); 之后，变量 SearchRec.Name 存放的便是“太空.scr”，当第一次运行代码

scrfiles := FindNext(SearchRec);之后，变量 SearchRec.Name 存放的便是“运动.scr”，接着不断循环直到找不到匹配的文件为止。



在菜单中动态关联过程



遇到难题

在清单 4-4 中讲述了如何动态创建菜单，同时对菜单的单击事件这一过程对象进行了赋值，如下列代码所述：

```
OurItem :=TMenuItem.Create(Self);
OurItem.Caption := SearchRec.Name;
OurItem.OnClick:=myClick;
```

通过这种方式可以实现菜单与过程的关联。但这种关联方式其实不利于程序的直观性以及可维护性。

另外，有时为了使用户的操作更方便，在程序设计中，除了菜单命令之外还创建了一些功能与菜单命令一样的按钮命令，而且希望按钮和菜单的标题以及小位图等都保持一致。在以往的 Delphi 版本中，为了实现这一功能需要单独对按钮和菜单的属性以及事件单独进行定义。

在 Delphi4 或 Delphi5 中，是否有更好更快捷的方法呢？



钥匙在此

Delphi4 和 Delphi5 中增加了 TactionList 控件，该控件可以很容易的解决以上两个问题。行为列表（Action lists）可以集中的管理一组响应用户操作控件或者菜单的命令。本节将介绍如何使用行为列表与目标对象或控件进行交互。

1. 行为对象（Action objects）

所谓行为，是指用户操作目标对象的命令，例如，当对编辑框中选中的高亮度文本进行拷贝时，该编辑框就是行为的目标对象，拷贝高亮度文本就是行为。可以在 TactionList 控件的编辑器中创建一组行为（Actions）。以下是行为/行为列表（action/action list）机理中每一组件类型的描述。

行为对象（TAction）是行为的实现部分，例如，拷贝编辑框中选中的文本。一个行为由响应用户命令（如单击鼠标）的客户组件所触发。典型的客户组件有菜单项和按钮。

行为列表组件（TactionList）是维护一系列行为对象的组件。该列表是设计程序时使用 Actions 的接口。

行为连接（TactionLink）是维护行为与客户组件的连接的对象。行为连接决定这些行为对于给定的客户组件是否可用或者哪一个行为可用。

典型的行为的客户组件是菜单项和按钮（ToolButton、TspeedButton、TmenuItem、Tbutton、TcheckBox、TradioButton 等）。一个行为被客户组件上的相应命令激发。典型的例子有，客户组件单击命令与行为的 Execute 方法相联系。

行为目标（An action target）通常是一个控件，例如，TrichText、Tmemo 或者数据控件等。

图 4-4 描述了这组对象的关系。在图中，Cut1 是行为，ActionList1 是载有 Cut1 的组件，SpeedButton1 则是 Cut1 的客户，Memo1 是目标。与行为对象、行为列表、行为客户以及行为目标不同，行为连接是非可见对象。行为连接在该图中是一个白色的矩形。这个行为连接使 SpeedButton1 客户与 ActionList1 中的 Cut1 行为联系在一起。

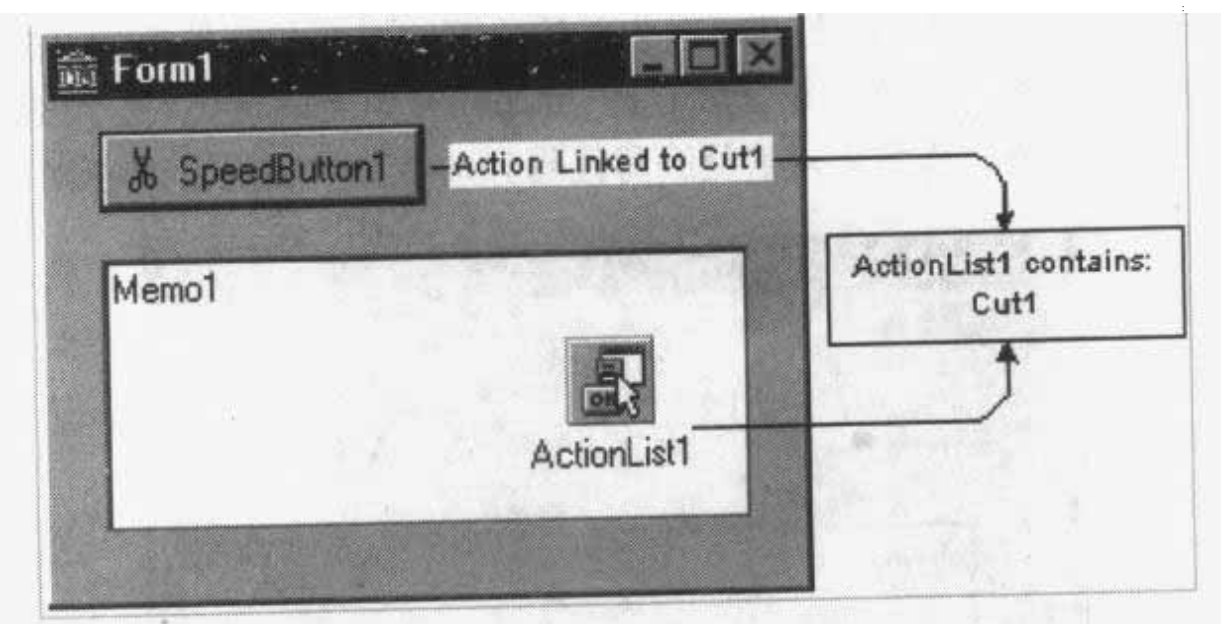


图 4-4　行为对象关系图

2. 使用行为（Actions）

从组件栏的标准页面中选择行为列表控件，并放置到窗体中或者数据模块（data module）中。双击该控件以显示行为列表编辑框，此时，可以添加、删除、或者重新排列行为列表的顺序。

在对象观察器（Object Inspector）中，可以设置每一行为的属性。Name 属性标识了行为，其他属性和事件（Caption、Checked、Enabled、HelpContext、Hint、ImageIndex、ShortCu以及 Visible）与客户控件的属性相对应。一般的，这些属性与客户组件的属性具有相同的名字，但这不是必然的。例如，行为的 Checked 属性与相应客户组件的 Down 属性就不具有相同的名字。

下面以一个例子来说明如何使用 TactionList 实现菜单项与过程的动态关联。



跟我来

1. 创建行为

新建一个应用程序，其设计状态下的主窗体如图 4-5 所示。其中，窗体名为 Form1，按钮控件名为 BitBtn1，菜单控件名为 MainMenu1，图像列表控件名为 ImageList1，行为列表控件名为 ActionList1，Tmemo 控件名为 memo1。

为 ImageList1 添加图像，然后设置 ActionList1.Images 属性为 ActionList1。在 ActionList1 添加行为 Cut，并设置属性如下：

```
Cut.ImageIndex:=0
Cut.Caption:='剪切'
```

2. 为客户控件设置 Action 属性

添加菜单项 N1，然后选择 N1 的 Action 属性为 Cut，并选择 BitBtn1 的 Action 属性也为 Cut。此时就会发现相应的按钮、菜单项的标题以及位图都变成了行为 Cut 所设置的属性值。



许多控件，如 TtoolButton、TspeedButton、TmenuItem 以及 Tbutton 等，都有 Action 属性。当使用行为列表控件添加了 Action 时，相应的行为对象的属性值将被拷贝到这些控件的 Action 属性中。此时，该行为对象所有的属性和事件都被这些控件所共享，并动态地与这些控件连接。因此，设置该行为（action）为无效状态时，相应的按钮和菜单项也自动变成了无效状态，从而无须重复编写代码分别使菜单和按钮无效。

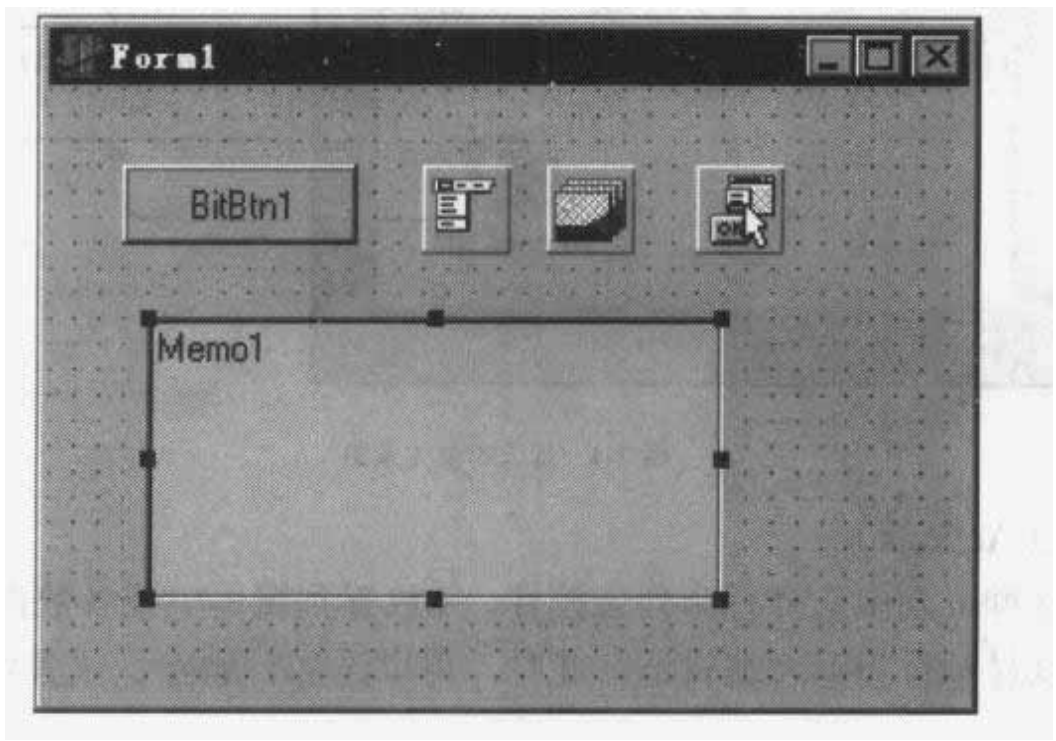


图 4-5　例子的设计界面

3. 为行为编写执行代码

当一个客户组件被单击时，与之相关联的行为的 OnExecute 事件被触发。例如，下列代码演示了“剪切”行为的 OnExecute 事件。

```
procedure TForm1.CutExecute(Sender: TObject);
```

```
begin
    Memo1.CutToClipboard;
end;
```

运行程序，单击按钮或者菜单项将把 Memo1 中所选中的文本剪切到剪贴板中。至此，已实现了菜单项与过程的动态关联。

第五章

控件的使用

热点透视

Delphi 的 VCL 已经对大量控件或者窗口组件进行了很好的封装，这些 VCL 构件的使用是如此的方便，作为程序员，您一定会有较深的体会吧。但是，有些内容并未得到封装，因此还需要程序员自己去加以完善。由于在 VCL 中要讲述的内容太多了，而 Delphi 已经提供了非常丰富的组件体系，因此，本章将只对程序员平时经常碰到的问题或者比较实用的内容进行讲述。本章内容有：

- 在运行时用鼠标移动控件
- 在 ListBox 和 ComboBox 中插入图片
- 为 ListBox 加入水平滚动条
- 截获滚动条的滚动消息
- 动态创建组件



在运行时用鼠标移动控件



遇到难题

您也许正在编写一个通用的应用程序，它允许用户定义自己的界面外观。如播放 MP3 的 Winamp 软件，它就允许用户自定义显示外观，只要用户所提供的界面定义格式满足它所要求的格式即可。也许您的界面还允许用户进行调整，如在程序运行时，允许用户对他所定义的界面控件进行移动等等。这在我们的程序中都能实现。当然，要实现在程序运行时完全由用户自定义界面，像 Delphi 的 IDE 一样，则程序还是比较庞大的，本节只是讲解其中的一小部分，即如何使自己的应用程序在运行时允许用户用鼠标移动窗体中的控件。



钥匙在此

为了移动控件，首先可以使用控件的 OnMouseDown 事件来记住鼠标的坐标位置，并使鼠标处于被捕获状态，然后使用 OnMouseMove 来跟踪鼠标的移动，移动控件直到控件的 OnMouseUp 事件发生。此时需要将控件放置到最终的地方并释放对鼠标的捕获。

要捕获鼠标需要调用 Win32 API 函数 SetCapture()。该函数用于捕获鼠标，所谓捕获鼠标，也就是说，此时的鼠标是属于当前线程的指定窗体的。一旦一个窗体捕获了鼠标，所有的鼠标输入都将被指引到该窗体中，而不管光标是否在该窗体的边界内。SetCapture（）函数的定义格式如下：

```
HWND SetCapture(
    HWND hWnd // 捕获鼠标的窗体句柄
);
```

当窗体不再需要鼠标输入时，产生窗体的线程调用 ReleaseCapture（）来释放捕获的鼠标。



跟我来

清单 5-1 显示了在窗体中如何使用鼠标移动按钮控件。用户在按钮上单击鼠标键的同时按住【Ctrl】键即可移动该按钮。

清单 5-1　移动按钮

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
TFrmMain = class(TForm)
    Button1: TButton;
    procedure Button1MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Button1MouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure Button1MouseUp(Sender: TObject; Button:
      TMouseButton; Shift: TShiftState; X, Y: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
    MouseDownSpot : TPoint;
    Capturing : bool;
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.Button1MouseDown(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if ssCtrl in Shift then begin  //当鼠标按下的同时按了【Ctrl】键
   //button1 捕获了鼠标
    SetCapture(Button1.Handle);
```

```
      Capturing := true;
      MouseDownSpot.X := x;
      MouseDownSpot.Y := Y;
    end;
end;

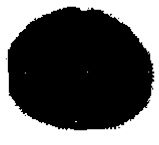
procedure TFrmMain.Button1MouseMove(Sender: TObject;
Shift: TShiftState; X,    Y: Integer);
begin
   if Capturing then begin
      Button1.Left := Button1.Left - (MouseDownSpot.x - x);
      Button1.Top := Button1.Top - (MouseDownSpot.y - y);
   end;
end;

procedure TFrmMain.Button1MouseUp(Sender: TObject; Button:
     TMouseButton; Shift: TShiftState; X,    Y: Integer);
begin
   if Capturing then begin
      ReleaseCapture;
      Capturing := false;
      //移动按钮至当前位置
      Button1.Left := Button1.Left - (MouseDownSpot.x - x);
      Button1.Top := Button1.Top - (MouseDownSpot.y - y);
   end;
end;

end.
```



在 ListBox 和 ComboBox 中插入图片



遇到难题

有时为了改变 ListBox 和 ComboBox 控件的显示外观，需要在 ListBox 和 ComboBox 中

插入图片，从而与字符串一起构成这些控件的列表项 Items，或者是直接使用图片来替代原先的字符串。这无疑给友好的界面效果又添加了一份姿色。其实 Delphi 已经为这两个控件留下一个实现这一功能的接口。



钥匙在此

对于 ListBox 和 Combobox 控件，如果 style 属性设为 csOwnerDrawFixed 或 csOwnerDrawVariable，那么在需要画列表时 OnDrawItem 事件就会发生。如果 style 属性为 csOwnerDrawVariable 时，OnMeasureItem 事件也会发生。而这两个事件为插入图片提供了接口。

1. 设置 style 属性

该属性用于确定这两个控件的显示类型。

对于 TcomboBox 而言，设置该属性，可以指定列表框和编辑框是如何出现的。该属性为一个枚举数据类型：

type TComboBoxStyle = (csDropDown , csSimple , csDropDownList , csOwnerDrawFixed, csOwnerDrawVariable);

如果 style 属性设为 csOwnerDrawFixed 或 csOwnerDrawVariable，那么在需要画列表时 OnDrawItem 事件就会发生。如果 style 属性为 csOwnerDrawVariable 时，OnMeasureItem 事件也会发生。

对于 TlistBox 而言，可以使用该属性来指定 Listbox 是一个标准的 listbox，还是一个允许显示图片的 owner-draw 类型的列表框，这种列表框允许以一个非标准的方式显示列表项。在这种情况下，必须编写代码。要创建一个 owner-draw 列表框，只需设置该属性值为 lbOwnerDrawFixed 或者 lbOwnerDrawVariable 即可。

2. OnDrawItem 事件

对于控件 ComboBox 而言，在 owner-draw 类型的组合框中，当一个显示项需要显示时激发该事件。这里使用一个事件句柄，在 owner-draw 组合框的下拉列表中绘制各项 Items。在该事件句柄中，可以加入图像元素至列表项中，或者使用图像替代列表项文本。

该事件句柄的类型为：

type TDrawItemEvent = procedure(Control: TWinControl; Index: Integer Rect: TRect;
State: TOwnerDrawState) of object;

可以在画布上使用 Rect 所提供的坐标位置来绘制各列表项。注意，只有当 style 属性被设置成 csOwnerDrawFixed 或者 csOwnerDrawVariable 该事件发生时。如果该事件并未进行定义，则组合框使用当前的刷子属性来填充 Rect，并写入各项的文本值。

对于 TlistBox 而言，当 owner-draw 列表框中的各项需要重画时激发该事件。该事件句柄的类型为：

type TDrawItemEvent = procedure(Control: TWinControl; Index: Integer Rect: TRect;
State: TOwnerDrawState) of object;

在该句柄中，参数 control 表示载有列表项控件的实例；参数 index 是当前列表项 Item 的索引；参数 Rect 是在该控件画布中列表项的坐标位置；参数 state 表示列表项 Item 的状态，

该状态意味着该项 Item 是否被选择、是否被检查或者是否聚焦等等，该属性的具体取值可以参见 Delphi 的在线帮助文件。

3. OnMeasureItem 事件

对于组合框而言，如果 style 属性为 csOwnerDrawVariable 的组合框中的项 Item 需要重画时，就会激发该事件。该事件的类型为：

type TMeasureItemEvent = procedure(Control: TWinControl; Index: Integer; var Height: Integer) of object;

property OnMeasureItem: TMeasureItemEvent;

如果组合框的 style 属性被设置成 csOwnerDrawVariable，那么 OnMeasureItem 事件先于 OnDrawItem 事件发生。通过使用该事件句柄，可以指定需要在列表框中绘制的 Item 的高度。OnMeasure Item 事件需要传递三个参数给该事件句柄。参数 control 表示载有 Item 组合框的实例；参数 Index 表示 Items 属性中的索引；参数 Height 表示该 Item 默认的高度。

对于列表框 ListBox 而言，当应用程序需要在一个可变高度的 owner-draw 列表框中重新显示各项 Item 时激发该事件。该事件类型为：

type TMeasureItemEvent = procedure(Control: TWinControl; Index: Integer; var Height: Integer) of object;

property OnMeasureItem: TMeasureItemEvent;

当 ListBox 的 style 属性设置为 lbOwnerDrawVariable 时，则使用 OnMeasureItem 事件写一个句柄来获得被画的各列表项的 items 属性值。该事件也有三个参数。Control 表示载有该项 Item 的控件；Index 指定该项在控件中的索引；参数 Height 指定该项 Item 的高度。



跟我来

现在看一个向 ListBox 和 ComboBox 中加入图片的例子，可以按照以下步骤进行。完整代码参见清单 5-2。

（1）创建一个新窗体；

（2）在窗体中放置一个 ComboBox 和 ListBox 控件；

（3）改变这些组件的 style 属性，将 ComboBox 的 style 属性设置成 csOwnerDrawVariable，将 ListBox 的属性设置成 lbOwnerDrawVariable。

一个自画的 TlistBox 和 TcomboBox 允许在 Items 属性中同时显示对象（例如图像）和字符串。例如，在下述例子中可以同时显示这些图像对象和字符串；

（4）在 Form 的变量中创建 5 个 Tbitmap 类型的变量；

（5）为窗体的创建事件创建一个过程；

（6）为 ComboBox 和 ListBox 的 OnDraw 事件创建一个过程；

（7）为 ComboBox 和 ListBox 的 OnMeasureItem 事件创建一个过程；

（8）在窗体的 OnClose 事件中释放资源。

清单 5-2　向 ListBox 和 ComboBox 中插入图片

```
unit OWNERDRW;
```

```
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    ComboBox1: TComboBox;
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
    procedure ComboBox1DrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure ComboBox1MeasureItem(Control: TWinControl; Index: Integer;
      var Height: Integer);
    procedure ListBox1DrawItem(Control: TWinControl; Index: Integer;
      Rect: TRect; State: TOwnerDrawState);
    procedure ListBox1MeasureItem(Control: TWinControl; Index: Integer;
      var Height: Integer);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  TheBitmap1, TheBitmap2, TheBitmap3, TheBitmap4,
  TheBitmap5 : TBitmap;
implementation

{$R *.DFM}

procedure TForm1.FormCreate(Sender: TObject);
begin
  TheBitmap1 := TBitmap.Create;
  TheBitmap1.LoadFromFile('D:\Borland\Borland
Shared\images\buttons\globe.bmp');
```

```
    TheBitmap2 := TBitmap.Create;
    TheBitmap2.LoadFromFile('D:\Borland\Borland
Shared\images\buttons\video.bmp');
    TheBitmap3 := TBitmap.Create;
    TheBitmap3.LoadFromFile('D:\Borland\Borland
Shared\images\buttons\gears.bmp');
    TheBitmap4 := TBitmap.Create;
    TheBitmap4.LoadFromFile('D:\Borland\Borland Shared\images\buttons\key.bmp');
    TheBitmap5 := TBitmap.Create;
    TheBitmap5.LoadFromFile('D:\Borland\Borland
Shared\images\buttons\tools.bmp');
    ComboBox1.Items.AddObject('Bitmap1: Globe',  TheBitmap1);
    ComboBox1.Items.AddObject('Bitmap2: Video',  TheBitmap2);
    ComboBox1.Items.AddObject('Bitmap3: Gears',  TheBitmap3);
    ComboBox1.Items.AddObject('Bitmap4: Key',  TheBitmap4);
    ComboBox1.Items.AddObject('Bitmap5: Tools',  TheBitmap5);
    ListBox1.Items.AddObject('Bitmap1: Globe',  TheBitmap1);
    ListBox1.Items.AddObject('Bitmap2: Video',  TheBitmap2);
    ListBox1.Items.AddObject('Bitmap3: Gears',  TheBitmap3);
    ListBox1.Items.AddObject('Bitmap4: Key',  TheBitmap4);
    ListBox1.Items.AddObject('Bitmap5: Tools',  TheBitmap5);

end;

procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    TheBitmap1.Free;
    TheBitmap2.Free;
    TheBitmap3.Free;
    TheBitmap4.Free;
    TheBitmap5.Free;
end;

procedure TForm1.ComboBox1DrawItem(Control: TWinControl; Index: Integer;
    Rect: TRect; State: TOwnerDrawState);
var
    Bitmap: TBitmap;
    Offset: Integer;
begin
    with (Control as TComboBox).Canvas do
```

```
begin
  FillRect(Rect);
  Bitmap := TBitmap(ComboBox1.Items.Objects[Index]);
  if Bitmap <> Nil then
  begin
    BrushCopy(Bounds(Rect.Left + 2,  Rect.Top + 2,  Bitmap.Width,
              Bitmap.Height),  Bitmap,  Bounds(0,  0,  Bitmap.Width,
              Bitmap.Height),  clRed);
    Offset := Bitmap.width + 8;
  end;
  { display the text }
  TextOut(Rect.Left + Offset,  Rect.Top,  Combobox1.Items[Index])
  end;
end;

procedure TForm1.ComboBox1MeasureItem(Control: TWinControl; Index:
                                                    Integer; var Height: Integer);
begin
  height:= 20;
end;

procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
var
  Bitmap: TBitmap;
  Offset: Integer;
begin
  with (Control as TListBox).Canvas do
begin
  FillRect(Rect);
  Bitmap := TBitmap(ListBox1.Items.Objects[Index]);
  if Bitmap <> Nil then
  begin
    BrushCopy(Bounds(Rect.Left + 2,  Rect.Top + 2,  Bitmap.Width,
              Bitmap.Height),  Bitmap,  Bounds(0,  0,  Bitmap.Width,
              Bitmap.Height),  clRed);
    Offset := Bitmap.width + 8;
  end;
  { display the text }
  TextOut(Rect.Left + Offset,  Rect.Top,  Listbox1.Items[Index])
```

```
    end;
end;

procedure TForm1.ListBox1MeasureItem(Control: TWinControl; Index: Integer;
    var Height: Integer);
begin
    height:= 20;
end;

end.
```

运行该程序，效果如图 5-1 所示。

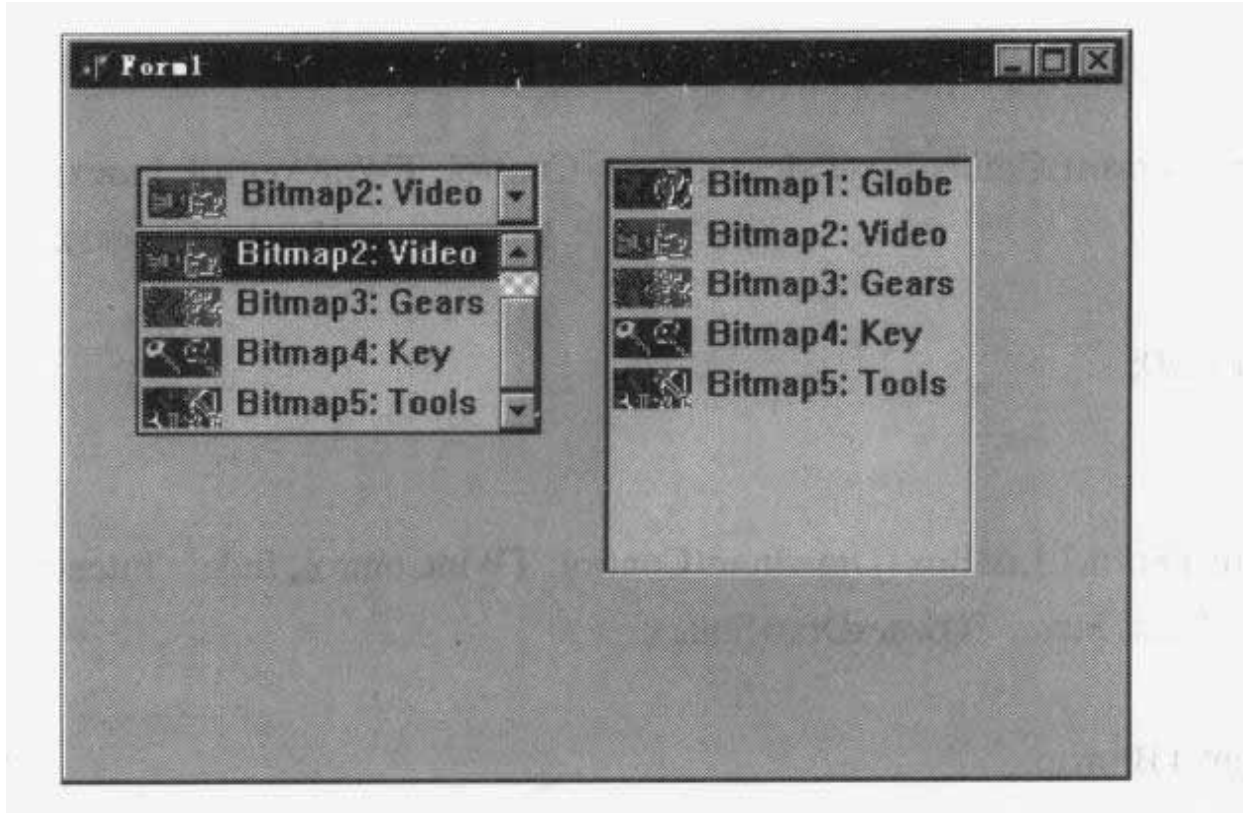


图 5-1　向 ListBox 和 ComboBox 中插入图片



为 ListBox 加入水平滚动条



遇到难题

Delphi 的 TListBox 元件自动实现了垂直滚动条，当 ListBox 的高度不能显示全部的列表项(Item)时，这个垂直滚动条就会出现。然而，当列表项的宽度比 ListBox 宽时，水平的滚动条却不会出现。



钥匙在此

其实实现这一功能并不难。通过调用 API 函数 SendMessage()向要横向滚动的列表框发送滚动的消息 LB_SETHORIZONTALEXTENT 即可解决这个问题。

一个应用程序发送的 LB_SETHORIZONTALEXTENT 消息可以设置列表框被水平滚动的宽度（即可滚动宽度）。如果列表框的宽度小于这个值的话，则水平滚动条将在列表框中出现，否则，水平滚动条将被隐藏。该消息常量相应的消息记录的两个参数在 API 中的声明分别为：

```
wParam = (WPARAM) cxExtent; // 水平滚动宽度
lParam = 0;                 // 不使用，必须设置为 0
```

例如，向 listbox1 发送该消息的例子代码为：

```
SendMessage(ListBox1.Handle， LB_SETHORIZONTALEXTENT， 100， 0);
```



跟我来

在 Form 的 OnCreate 事件中添加以下代码：

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i， MaxWidth: integer;
begin
  MaxWidth := 0;
  for i := 0 to ListBox1.Items.Count - 1 do
  if MaxWidth < ListBox1.Canvas.TextWidth(ListBox1.Items.Strings[i]) then
    MaxWidth := ListBox1.Canvas.TextWidth(ListBox1.Items.Strings[i]);
  SendMessage(ListBox1.Handle， LB_SETHORIZONTALEXTENT， MaxWidth+2， 0);
  end;
```

以上的代码首先得到 ListBox 中最长 Item 的长度，然后用 LB_SETHORIZONTALEXTENT 消息设置 ListBox 的水平滚动的宽度，宽度加 2 是为了给右面留 2 个点的边距。运行该程序代码，如果 Item 文本内容宽度超过 ListBox 原有的宽度的话，将显示水平滚动条，结果如图 5-2 所示。



图 5-2 ListBox 的水平滚动条



截获滚动条的滚动消息



遇到难题

Delphi 的 VCL 已经对大量控件或者窗口组件进行了很好的封装，但是，所封装的事件并未全部覆盖控件应该进行处理的消息，例如，对于 Tscrollbox、TDBGrid 等等控件中滚动条的滚动消息就未提供相应的事件。虽然，Delphi 已经对其进行了缺省处理，但有时程序员需要截获一些消息，而相应事件过程并未提供的话，此时就需要进行编程来处理了。



钥匙在此

如何截获消息，这在消息系统章节中已经详细介绍过了，请参见相应的章节。若要截获滚动条滚动的消息，则还应该了解滚动消息的名称以及参数的意义。

当滚动条控件或者一些控件上的滚动条发生滚动时，如果是水平滚动条，则会向系统发送 WM_HSCROLL 消息；如果是垂直滚动条，则会向系统发送 WM_VSCROLL 消息。

WM_HSCROLL 消息以及 WM_VSCROLL 消息相应的消息记录中的参数在 API 中是这样声明的：

```
nScrollCode = (int) LOWORD(wParam);   // 滚动 bar 的值
nPos = (short int) HIWORD(wParam);    // 滚动 box 的位置
hwndScrollBar = (HWND) lParam;        // 滚动 bar 的句柄
```

另外，API 函数也可以获取或者设置当前滚动条的一些信息。如，滚动 Box 的最大或最小值，以及滚动 Bar 的当前位置等等。

例如，获取滚动 Box 的最大最小值的 API 函数有 GetScrollRange()，相应的设置函数为 SetScrollRange()。GetScrollPos() 函数可以得到当前滚动 box 的位置，SetScrollPos()函数可以设置当前滚动 Box 的位置。



跟我来

以下例子显示了如何截获 Tscrollbox 构件的滚动消息，并保持两个滚动条的同步。如果一个滚动条发生滚动，那么另一个滚动条的位置也改成同样的值。滚动消息被滚动 box 的 WinPorc（）所捕获。清单 5-3 为该例子的完整代码。

清单 5-3　截获滚动条的滚动消息

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TForm1 = class(TForm)
    ScrollBox1: TScrollBox;
    Label1: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
type
{$IFDEF WIN32}
  WParameter = LongInt;
{$ELSE}
  WParameter = Word;
{$ENDIF}
  LParameter = LongInt;

{Declare a variable to hold the window procedure we are replacing}
var
  OldWindowProc : Pointer;
```

```
function NewWindowProc(WindowHandle : hWnd;
                       TheMessage    : WParameter;
                       ParamW        : WParameter;
                       ParamL        : LParameter) : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
var
  TheRangeMin : integer;
  TheRangeMax : integer;
  TheRange : integer;
begin

  if TheMessage = WM_VSCROLL then begin
{获取水平滚动条的最大、最小值}
  GetScrollRange(WindowHandle,
                 SB_HORZ,
                 TheRangeMin,
                 TheRangeMax);
{获得滚动条的垂直位置}
  TheRange := GetScrollPos(WindowHandle,
                           SB_VERT);
{确保不越过规定范围}
  if TheRange < TheRangeMin then
    TheRange := TheRangeMin else
  if TheRange > TheRangeMax then
    TheRange := TheRangeMax;
{设置水平滚动条的位置}
  SetScrollPos(WindowHandle,
               SB_HORZ,
               TheRange,
               true);
end;

if TheMessage = WM_HSCROLL then begin
{获得垂直滚动条的最大和最小范围}
  GetScrollRange(WindowHandle,
                 SB_VERT,
                 TheRangeMin,
                 TheRangeMax);
{获取滚动条的水平位置}
  TheRange := GetScrollPos(WindowHandle,
```

```
                                        SB_HORZ);
  {确保不越过规定范围}
    if TheRange < TheRangeMin then
      TheRange := TheRangeMin else
    if TheRange > TheRangeMax then
      TheRange := TheRangeMax;
  {设置垂直滚动条的位置}
    SetScrollPos(WindowHandle,
                 SB_VERT,
                 TheRange,
                 true);
end;

{ 调用以前的窗口过程用于处理消息}
  NewWindowProc := CallWindowProc(OldWindowProc,
                                  WindowHandle,
                                  TheMessage,
                                  ParamW,
                                  ParamL);
end;


procedure TForm1.FormCreate(Sender: TObject);
begin
{ 为控件创建一个新的窗体过程并且记住老的窗体过程 }
  OldWindowProc := Pointer(SetWindowLong(ScrollBox1.Handle,
                                         GWL_WNDPROC,
                                         LongInt(@NewWindowProc)));
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
{ 将窗体过程设为老的窗体过程 }
  SetWindowLong(ScrollBox1.Handle,
                GWL_WNDPROC,
                LongInt(OldWindowProc));

end;

end.
```



动态创建组件



遇到难题

当前，有许多通用型的 MIS 软件支持用户的二次开发，例如，录入界面可以由用户自己创建等。清华紫光的档案管理系统就是一个产品化的通用软件，它具有动态创建录入界面的功能，使用户可以根据自己的需要对录入界面进行设计，当然，该系统中还包括了对数据库的动态创建。如何动态创建组件并存入文件中呢？



钥匙在此

1. 动态创建组件

动态创建组件还是比较容易的。Delphi 的各个类中都提供了 Create 的方法，将这一方法放入代码中就可以实现动态创建组件的功能。创建了这些组件后，不要忘了为创建的实例的属性进行设置，因为创建组件后，系统赋给它的属性都是缺省的值。例如，在将创建的可视组件放置到窗体上时，不要忘了设置该组件的 Parent 属性。

下面举一个例子，显示组件是如何动态创建的。这个例子需要一个 TpageControl 和 一个 TcheckBox 控件。在窗体的 OnCreate 事件句柄中为 TpageControl 创建了 sheet，同时在每一个 sheet 上又创建了编辑框。单击 CheckBox 将使 Tab 控件变成不可视。

```
procedure TForm1.PageControl1Change(Sender: TObject);

begin
  CheckBox1.Checked := PageControl1.ActivePage.Visible;
end;

procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  PageControl1.ActivePage.Visible := CheckBox1.Checked;
end;

procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
```

```
begin
  for i := 0 to 9 do
    with TTabSheet.Create(Self) do
    begin
      PageControl := PageControl1;

      Caption := 'TabSheet' + IntToStr(i);
      with TEdit.Create(Self) do
      begin
        Parent := PageControl1.Pages[i];
        Left := Random(PageControl1.ActivePage.ClientWidth - Width);
        Top := Random(PageControl1.ActivePage.ClientHeight - Height);
      end;
    end;
    PageControl1Change(Sender);
end;
```

2. 存储和装载控件信息

使用 TStream 可以使应用程序从磁盘文件、内存以及数据库的 Blob 字段中读取控件信息，也可以在它们中写入控件信息。TStveam 可以打开一个已命名的文件并且提供了读写文件的能力。

TStream.WriteComponent 方法可以将组件及其属性信息写入流文件。该方法用于 Delphi 组件的流系统中，当然，该方法也可以将组件及其属性直接写入内存中或者数据库的 Blob 字段中。其声明格式如下：

```
procedure WriteComponent(Instance: TComponent);
```

参数 Instance 即为要写入流文件中的组件实例。

TStream.ReadComponent 方法可以从流中读取数据，并将之赋给实例 Instance 的属性。该函数将返回一个组件。如果 Instance 为 Nil 时，该方法将在流中构造一个基于类型信息的组件，然后返回这个新构造的组件。

当然，在进行读写数据流之前，必须创建一个流实例或者流对象。对于不同的流方式，其所创建的函数的参数并不一样。例如，对于文件流，其说明格式如下：

```
constructor TFileStream.Create(const FileName: string; Mode: Word);
```

其中参数 FileName 为要创建的流文件的文件名；参数 Mode 说明了文件流如何被打开。例如，参数 Mode 取 fmCreate 表示创建一个新给定的文件；取 fmOpenRead 表示打开文件准备进行读取；取 fmOpenWrite 表示可以打开文件以及写文件。

下面例子显示了如何使用 VCL 组件的流系统来将一个组件转换成字符串，然后又将这个字符串转换成组件。

```
function ComponentToString(Component: TComponent): string;
var
  BinStream:TMemoryStream;
```

```
    StrStream: TStringStream;
    s: string;
begin
    BinStream := TMemoryStream.Create;
    try
        StrStream := TStringStream.Create(s);
        try
            BinStream.WriteComponent(Component);
            BinStream.Seek(0，  soFromBeginning);
            ObjectBinaryToText(BinStream，  StrStream);
            StrStream.Seek(0，  soFromBeginning);
            Result:= StrStream.DataString;
        finally
            StrStream.Free;

        end;
    finally
        BinStream.Free
    end;
end;

function StringToComponent(Value: string): TComponent;
var
    StrStream:TStringStream;
    BinStream: TMemoryStream;
begin
    StrStream := TStringStream.Create(Value);
    try
        BinStream := TMemoryStream.Create;
        try
            ObjectTextToBinary(StrStream，  BinStream);
            BinStream.Seek(0，  soFromBeginning);
            Result := BinStream.ReadComponent(Nil);

        finally
            BinStream.Free;
        end;
    finally
        StrStream.Free;
    end;
```

```
end;
```

3. 在拥有者组件上插入其他组件

在拥有者组件上插入其他组件只需调用该拥有者的 InsertComponent()方法。该方法声明如下：

```
procedure TComponent.InsertComponent(AComponent: TComponent);
```

该方法的作用就是将指定的 AComponent 插入到当前的组件中。下列代码演示了如何移走和插入组件。

```
var
  I: Integer;
  Temp: TComponent;
begin
  for I := ComponentCount - 1 downto 0 do
  begin
    Temp := Components[I];
    if not (Temp is TControl) then
    begin
      RemoveComponent(Temp);
      DataModule2.InsertComponent(Temp);
    end;
  end;
end;
```

4. 注册组件类

如果没有注册类的话，是不能使用 Delphi 的 VCL 流系统进行控件信息的读和写的。为了注册这些要存储或者读取的组件，必须调用 RegisterClasses（）或者 RegisterClass（）进行类的注册。前者可以注册一组类，后者只是注册一个类。以下代码，注册了 Tedit 以及 Tlabel 类。

```
registerClasses([TEdit，TLabel]);
```



跟我来

清单 5-4 是一个在动态创建组件的同时又将其存储起来的完整例子。新建一个应用程序，然后在窗体上放置一个 Tpanel 和两个 Tbutton 组件。

研究一下该程序，看它是如何动态创建组件，又如何将其存储到文件中，同时又如何将组件信息从文件中取出来，并将之显示出来的。

清单 5-4　动态创建组件

```
unit Unit1;
```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    SaveBtn: TButton;
    LoadBtn: TButton;
    Panel1: TPanel;
    procedure SaveBtnClick(Sender: TObject);
    procedure LoadBtnClick(Sender: TObject);
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure FormShow(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    NumEdits:integer;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

//下面的过程保存 Form 上的 Edit 组件
procedure TForm1.SaveBtnClick(Sender: TObject);
var
  AStream:TFileStream;
  I:integer;
begin
  AStream:=TFileStream.Create('Sample.Dat', fmCreate);
  for I := 0 to ControlCount - 1 do
   if ((Controls[I] is TEdit)or ((Controls[I] is TLabel))) then
      AStream.WriteComponent (Controls[I]);
```

```
  AStream.Free;
end;

 //下面的过程从文件中取出保存的结果
procedure TForm1.LoadBtnClick(Sender: TObject);       //Load your components
var
  AStream:TFileStream;
  AReader:TREader;
  New:TComponent;
begin
  AStream:=TFileStream.Create('Sample.Dat', fmOpenRead);
  while AStream.Position < AStream.Size do
  begin
    AReader := TReader.Create (AStream, 4096);
    try
      //表示读取的组件的拥有者为 panel1
      AReader.Parent := panel1;
      //从文件流中读取已经注册的组件
      New := AReader.ReadRootComponent (Nil);
      //在 panel1 上插入组件
      Panel1.InsertComponent (New);
    finally
      AReader.Free;
    end;
  end;
  AStream.Free;
end;

//动态创建组件
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //在鼠标的指定位置上动态创建组件
  with TLabel.Create(Self) do
  begin
    Left:=X;
    Top:=Y;
    Parent:=Self;
    Name:='Label'+IntToStr(NumEdits);
  end;
```

```
    with TEdit.Create(Self) do
    begin
      Left:=X+50;
      Top:=Y;
      Parent:=Self;
      Name:='Edit'+IntToStr(NumEdits);
    end;
    inc(NumEdits);
end;

procedure TForm1.FormShow(Sender: TObject);
begin
    NumEdits:=0;
end;

//下面的语句是千万不能少的
initialization
    registerClasses([TEdit，TLabel]);
end.
```

运行该代码后，当鼠标点击窗体时，将在鼠标点击的位置上动态创建一个标签以及一个编辑框。如果不断的点击就会不断的创建。单击“SaveBtn”按钮，则会将当前窗体上的所有的标签和编辑框都存储到 sample.dat 流文件中。单击“LoadBtn”按钮，则会将 sample.dat 数据读取出来，并将所读取的组件显示在 Panel1 上。运行该程序，点击窗体使之动态创建几个组件，然后按“SaveBtn”，将组件信息存储起来，再按“LoadBtn”按钮，又将这些组件在 Panel1 上显示出来。运行结果见图 5-3 所示。

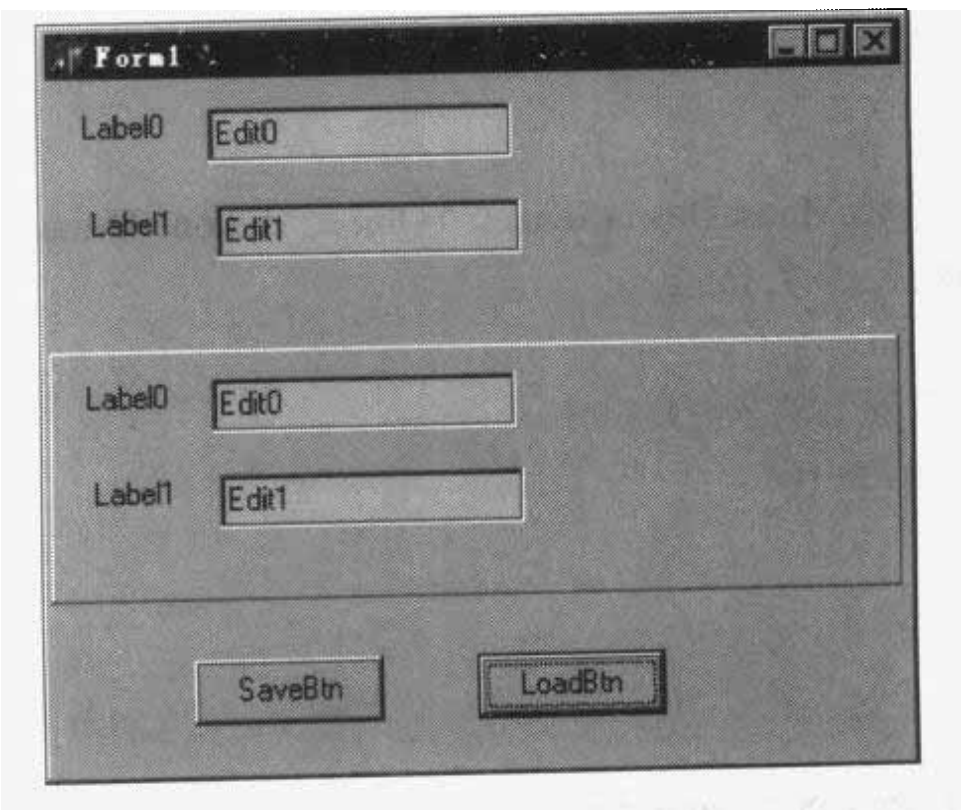


图 5-3　动态创建组件

第六章

资源文件

热点透视

如果不想让数据文件（如 wav、avi、bmp、cur 等多媒体数据文件）独立于应用程序而存在，而是将其包含到应用程序中则需要用到资源文件。资源文件包括了程序中所需要的资源数据信息。Delphi 中提供了使用常用资源（如位图、图标等）的标准函数，但如果资源数据为非标准资源文件时则需要调用系统函数。本章将选择一些典型的资源文件的使用方法进行阐述，包括：

- ✧ 自定义和使用资源文件
- ✧ 从资源文件中装载位图和光标
- ✧ 将 wav 文件包含到应用程序中
- ✧ 使用其他 DLL 或 EXE 文件中的资源文件



自定义和使用资源文件



遇到难题

编程中读者也许已经注意到，在项目文件.dpr 中缺省的加上了这样一条编译指令：

{$R *.RES}

该指令表明了在自己的应用程序中或者 DLL 文件中将包含指定的资源文件，*符号在编译指令中代表一个特殊的意义，它表示指令所在源代码文件的基名（即没带扩展名的文件名）。通常情况下，应用程序的资源文件（*.RES）与它的工程文件（*.DPR）有相同的名字，同样，窗体文件（*.DFM）与它的单元文件（*.PAS）有相同的名字，因此，在单元文件（*.PAS）中相应的会有{$R *.DFM}指令。

资源被编译到一个叫做“.RES”的文件中，该文件在创建或编译应用程序时将赋值到相应的应用程序中。即使在分发应用程序给用户后，应用程序所载的资源文件还是可以通过资源文件编辑器进行修改和编辑的。我们经常看到的很多汉化软件，一般都是修改应用程序的资源文件。Borland 的资源编辑器是随着开发工具一起提供的。这些开发工具可以修改 16 位和 32 位资源文件，无论资源文件是自带的、独立的或者是嵌入到.DLL 或.EXE 中的，都是所见即所得（WYSIWYG）的。如何自定义资源和使用资源呢？



钥匙在此

1. 创建资源文件

在 Delphi 中创建资源文件的方式一般有两种。

一种是直接创建方式。这种方式即直接使用资源开发工具来创建资源文件。Delphi 中提供了一个叫做 Image Editor 的工具，可以用来直接创建资源文件，在文件菜单中，选择“NewlResouce File”命令便可使用该工具。但通过这种方式只能在该资源文件中添加位图、图标和光标三种该工具能创建的资源。至于如何使用该工具本书不作详细介绍。

另一种是间接方法，即先创建一个资源定义文本文件（.RC），然后通过资源编译器将该文件编译成资源文件。

首先在应用程序所在目录中必须创建一个载有资源定义的文本文件。可以任意的对该文件进行命名，但其后缀名必须为“.rc”，并且文件基名与任何单元文件名或工程文件名都不相同。这在 Delphi 中是非常重要的，因为 Delphi 会为工程自动创建很多资源文件。例如，下面就是.rc 文件中的内容：

MYUSERDATA MYDATATYPE TEST.TXT

其中，MYUSERDATA 是资源名，MYDATATYPE 是资源的类型，TEST.TXT 是要连入程序中的所定义的载有数据的文件名。

接着，我们必须创建数据文件。本例为 TEST.TXT 文件，这可以使用任何一个文本编辑器来实现。在本例中，该文件中写入下列一行文字：

Hello!

要将.RC 文件编译成要连入应用程序的资源文件.RES，可以在 DOS 命令行中敲入以下命令行，在命令行中应当指明编译程序以及.RC 文件的路径名，例如：

c:\Delphi\Bin\brcc32.exe c:\Delphi\MYRES.RC

当编译器编译完后，应该看到一个与.RC 具有相同文件名并且已经编译过的资源文件，它的后缀名为“.RES”。

2. 将资源文件连入应用程序中

现在可以将资源文件连接到应用程序中了。在应用程序中简单的敲入以下语句，只不过实际使用时要将 ResFileName 替换成您自己的资源文件名。

{$R ResFileName.RES}

这里，ResFileName.RES 就是已经编译过的资源文件名。一旦资源文件被连接到应用程序中，就可以用任何一种模式来装载该资源文件。

3. 装载和使用资源文件

在实际使用资源时，需要调用几个 Windows API 函数。

首先，需要调用 FindResource（）函数，这需要将应用程序的句柄、要装载的资源的名称以及资源的类型传给该函数。如果执行成功，则函数返回该资源的句柄。

接着，如果需要知道所装载资源块在内存中的大小的话，可以调用 SizeOfResource()函数获取有关信息。需要注意的是，这不是数据的真实大小，而是装载资源时所分配的内存块的大小。如果需要知道数据的真实大小，则需要将数据的大小信息嵌入到数据中。

如果要装载资源，可以调用 LoadResource()函数，并将调用 FindResource（）函数时返回的资源句柄传给该函数。该函数可以返回所装载资源的句柄，以便传给 LockResource（）函数来获取指向资源数据的指针。

最后调用 UnLockResource（）函数和 FreeResource（）函数来释放资源。

在我们的例子中，我们将资源数据（上述例子为文本文件的内容）加入到一个字符串中，直到找到嵌入的“!”为止。



跟我来

在创建程序之前，首先应创建两个文件，即按照上述方法分别创建数据文件 test.txt 和资源定义文件 myres.rc，然后将资源定义文件编译成资源文件 myres.res。

源程序见清单 6-1 所示。该程序将文本数据从资源文件中输出，直到碰到符号“!”为止。单击按钮时，则弹出“Hello”对话框。

程序清单 6-1　使用资源文件

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
//包含自定义的资源文件
{$R MYRES.RES}

procedure TForm1.Button1Click(Sender: TObject);
var
  hRes          : THandle; {资源句柄}
```

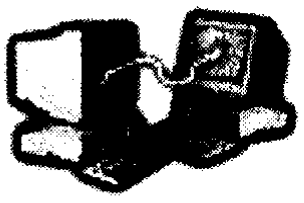
```
  pRes          : pointer; {资源指针}
  ResSize       : longint; {资源大小}
  i             : integer; {计数器}
 {$IFDEF WIN32}
  s             : shortstring; {a string to play with}
 {$ELSE}
  s             : string;      {a string to play with}
 {$ENDIF}
begin
 {查找资源}
  hRes := FindResource(hInstance,
                       'MYUSERDATA',
                       'MYDATATYPE');
 if hRes = 0 then begin
    ShowMessage('Could not find the resource');
    exit;
 end;
{获取资源在内存中的大小}
 ResSize := SizeOfResource(hinstance, hRes);
 if ResSize = 0 then begin
    ShowMessage('Nothing to load - size = 0');
    Exit;
 end;
{装载资源}
 hRes := LoadResource(hInstance, hRes);
 if hRes = 0 then begin
    ShowMessage('Resource Load Failure');
    Exit;
 end;
{获得资源指针}
 pRes := LockResource(hRes);
 if pRes = Nil then begin
    ShowMessage('Resource Lock Failure');
    FreeResource(hRes);
    Exit;
 end;

{将资源指针转换成字符串}
  s:='';
  i := 0;
```

```
    while pChar(pRes)[i] <> '!' do begin
      s := s + pChar(pRes)[i];
      inc(i);
    end;

  {prove it works}
    ShowMessage(s);

  {解锁和释放资源}
    UnLockResource(hRes);
    FreeResource(hRes);
end;
end.
```



从资源文件中装载位图和光标



遇到难题

从上一节中我们已经知道，可以通过资源文件储存自己程序中所需要的资源信息，并供程序调用。在我们的程序中，有时会使用一些位图以及自定义的光标。此时需要将这些信息存入资源文件中，并使这些资源连接到自己的程序里，而不是将一堆资源文件置于应用程序之外。如何实现这一功能呢？



钥匙在此

位图以及光标都可以存储在资源文件.RES 中，并可连接到应用程序.EXE 中。这些资源文件可以使用 Delphi 的 Image Editor 或者 Borland 的 Resource WorkShop 工具来创建。在连入应用程序之后，储存在资源文件中的位图和光标可以分别通过调用 API 函数 LoadBitmap（）和 LoadCursor（）来获取。

1. 装载位图

LoadBitmap（）API 函数定义如下：

```
function LoadBitmap(Instance: THandle;
                    BitmapName: PChar): HBitmap;
```

第一个参数获取资源文件的 EXE 或者 DLL 的实例句柄。Delphi 中提供了 EXE 的实例句柄，该句柄是一个称为 Hinstance 的全局变量。以下这个例子从应用程序中装载位图。然而，载有资源文件的应用程序也可以是另一个 EXE 或者 DLL 文件。以下这个例子从应用程序的资源文件中装载一个称为 BITMAP_1 的位图。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Bmp: TBitmap;
begin
  Bmp := TBitmap.Create;
  Bmp.Handle := LoadBitmap(HInstance,'BITMAP_1');
  Canvas.Draw(0, 0, Bmp);
  Bmp.Free;
end;
```

2. 装载光标

装载光标函数 LoadCursor（）是如下定义的：

```
function LoadCursor(Instance: THandle;
                    CursorName: PChar): HCursor;
```

第一个参数是载有资源文件的应用程序实例句柄变量。与上面一样，下列这个例子从应用程序中进行装载光标。第二个参数是光标的名称。

首先在接口处声明：

```
const
  crMyCursor = 5; {Other units can use this constant}
```

接着，将下列两行代码放入窗体创建的事件中。

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Screen.Cursors[crMyCursor] := LoadCursor(HInstance, 'CURSOR_1');
    Cursor := crMyCursor;
end;
```

可以按照下列例子改变 Delphi 的标准光标（光标常量可以在在线帮助中的 Cursors 属性中找到）。



正常情况下，使用 DeleteCursor 将删除任何一个指定的光标资源，然而，Delphi 也可以在 Cursors 数组中删除所有光标。

下面可以看一个完整的例子，看是如何从资源文件中获取位图和光标的。



跟我来

从资源文件中获取位图和光标的源程序见清单 6-2。

清单 6-2　从资源文件中获取位图和光标

```
implementation

{$R *.DFM}
 //包含了资源文件 myres.RES
{$R myres.RES}

const
   crMyCursor = 5;

procedure TForm1.Button1Click(Sender: TObject);
var
   Bmp: TBitmap;
begin
   Bmp := TBitmap.Create;
     //获得所定义的位图
   Bmp.Handle := LoadBitmap(HInstance,'BITMAP_1');
   Canvas.Draw(0, 0, Bmp);
   Bmp.Free;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
   //获取所定义的光标
   Screen.Cursors[crMyCursor] := LoadCursor(HInstance, 'CURSOR_1');
   //将所获取的光标设成当前光标
   Cursor := crMyCursor;

end;
```

运行该程序，单击窗体中的画图按钮，其运行结果如图 6-1 所示。同时在光标移动到该窗体的区域中时光标也会发生变化，变成您所定义的光标，其中图中已经标明了图表的图形样式。

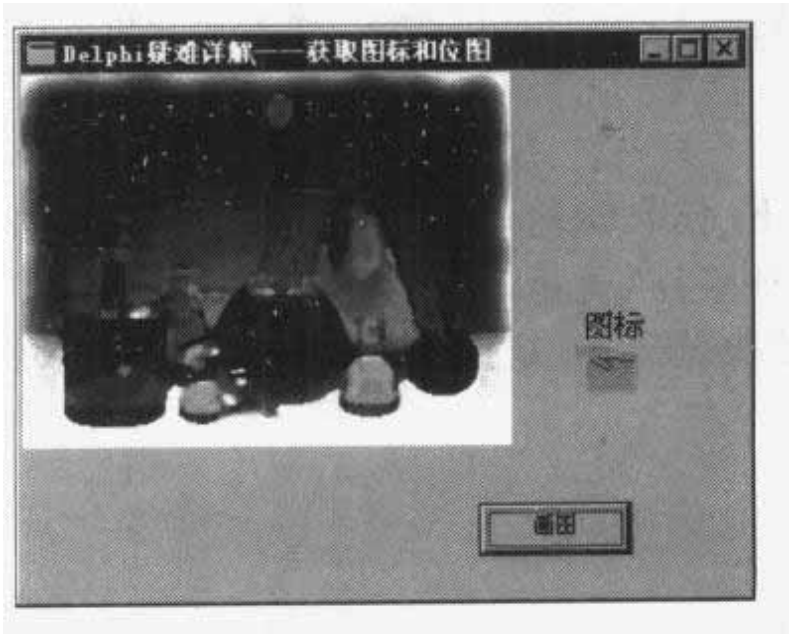


图 6-1　获取自定义的位图和光标



将 wav 文件包含到应用程序中



遇到难题

现在您也许想将多媒体数据文件，例如 wav 文件包含入您的应用程序中，但又不希望让这些数据文件独立于应用程序而存在。也就是说，要将这些多媒体数据文件包含到应用程序中，同时又能在应用程序中调用它们。如何实现这一功能呢？



钥匙在此

通过阅读第一节的内容，相信读者对如何在程序中使用资源文件已经有了一定的认识。对于本节而言，要将多媒体数据文件，例如 wav 文件，也包含到应用程序中，同样可以通过将这些文件包含到资源文件中来实现。



跟我来

现在就以两个 wav 文件为例，将文件 1.wav，2.wav 包含到应用程序中，并将这两个 wav 文件播放出来。

首先，生成一个 foo.rc 文件，这是一个纯文本文件。foo.rc 文件中包含的内容为，

```
WAVE1 WAVE PRELOAD FIXED PURE "1.WAV"
WAVE2 WAVE PRELOAD FIXED PURE "2.WAV"
```

这里第一个单词是资源名，这些资源名必须记住；第二个单词为资源的类型，允许是自定义的类型；接着的两个单词为资源关键字选项，可以不包含这些关键字，PRELOAD 表示立即装入资源，FIXED 表示保持资源在固定的内存位置；最后一个单词为 wav 文件名。这里，最重要的三个参数为资源名、资源类型以及资源的文件名。

接着将资源的定义文件编译成资源文件，本例的编译命令为：

```
c:\Delphi\Bin\brcc32.exe c:\Delphi\foo.rc
```

此时形成了资源文件 foo.res。最后写入以下代码，如清单 6-3 所示。

清单 6-3　使用 wav 资源

```
interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls,mmSYSTEM;//为了能够播放波表文件，需要使用 mmSYSTEM 单元文件

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
//连接资源文件
{$R foo.res}
var
  rhMyNoise : THandle;
  pMyNoise : Pointer;
  hMyNoise : THandle;
procedure TForm1.Button1Click(Sender: TObject);
begin
  //查找资源
  rhMyNoise := FindResource(HInstance, 'WAVE1', 'WAVE');
//获取资源句柄
  hMyNoise := LoadResource(HInstance, rhMyNoise);
  //获取资源数据的指针
pMyNoise := LockResource(hMyNoise);
//使用资源数据，本例为播放波表文件
  sndPlaySound(pMyNoise, SND_SYNC or SND_MEMORY);
//释放资源
```

```
    FreeResource(hMyNoise);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    rhMyNoise := FindResource(HInstance, 'WAVE2', 'WAVE');
    hMyNoise := LoadResource(HInstance, rhMyNoise);
    pMyNoise := LockResource(hMyNoise);
    sndPlaySound(pMyNoise, SND_SYNC or SND_MEMORY);
    FreeResource(hMyNoise);

end;
end.
```

该代码中 sndPlaySound（）为播放波表文件的系统函数，第一个参数表示播放的声音的标识字符串，该参数既可以是在注册表中标识系统声音的标识符，也可以是波表文件名；第二个参数为播放标志位，本例使用了 SND_SYNC 和 SND_MEMORY 常量。其中，常量 SND_SYNC 表示同步播放声音，并且播放完后，函数才返回；常量 SND_MEMORY 表示参数一为指向内存中声音数据的指针。参数二还可以取其他常量，具体的含义可以参见 Win32 API 说明。

该程序运行时界面如图 6-2 所示，当单击按钮时，如果音箱打开的话，将可以听到优美的波表文件。

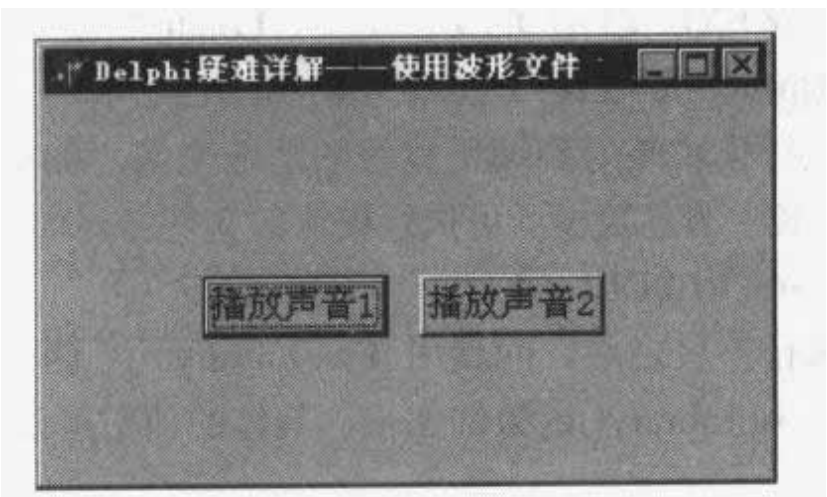


图 6-2　播放波表文件资源



再送礼包

我们还可以使用 Borland 所提供的资源编辑器来将自定义的资源（本例为波表文件）直接连接到资源文件中，而不需先书写并编译资源定义文件，这样可以省去一些麻烦。

对于其他的多媒体文件，如 avi 文件等，其使用方法与本例类似。



使用其他 DLL 或 EXE 文件中的资源文件



遇到难题

在前三节例子中所有装载的资源都是来自于同一个应用程序，即资源文件包含在调用它的应用程序中。如 FindResource（）以及 LoadResource（）等函数使用的应用程序句柄都是 Hinstance，该变量缺省的值是系统为本程序分配的句柄，所以，这样所使用的资源都是本应用程序所包含的资源文件。但程序员有时往往要使用来自于另一个应用程序（.EXE）或动态连接库（.DLL）的资源，这时为了使用资源文件，应使用其他函数来获得所使用的应用程序或者动态连接库的句柄。



钥匙在此

为了使用其他 DLL 或者 EXE 文件中的资源文件，需调用 FindResource()以及 LoadResource（）等函数，在此之前应当先调用 LoadLibrary（）或者 LoadLibraryEx（）函数获取可执行模块（DLL 或 EXE 文件）的文件句柄。LoadLibrary()的定义文法如下：

```
function LoadLibrary(lpLibFileName: PChar): HMODULE;
```

该函数将指定的可执行模块映射到调用线程的地址空间，如果调用成功的话，返回值为该可执行模块的句柄。其中的参数表示可执行模块的文件名。如：

```
myhandle := LoadLibrary('RESDLL.DLL');
```

当使用完相应的可执行模块之后，应调用 FreeLibrary（）函数来释放所占用的资源。LoadLibraryEx（）是对 LoadLibrary()函数的加强，具体的使用可以参见 Win32 API 的函数调用的说明。

下面看一个从动态连接库中获取位图的例子。



跟我来

先创建一个空的 DLL 工程，该工程文件中连接了载有资源信息的资源文件，本例中使用的资源文件是第二节中使用的资源文件。工程文件的源代码见清单 6-4。

清单 6-4　载有资源文件的 DLL 工程文件

```
library resdll;
```

```
uses
   SysUtils;
//连接资源文件
{$R myres.RES}
begin
end.
```

为了使用 DLL 中的资源，可以在应用程序中加入以下代码，程序代码见清单 6-5。

清单 6-5　使用 DLL 中的资源

```
implementation

{$R *.DFM}

{$IFDEF WIN32}
const BadDllLoad = 0;
{$ELSE}
const BadDllLoad = 32;
{$ENDIF}

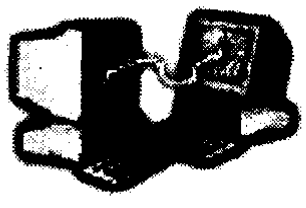
procedure TForm1.Button1Click(Sender: TObject);
var
   h : THandle;
   Bmp: TBitmap;
begin
   //获得 DLL 的句柄
   h := LoadLibrary('RESDLL.DLL');
   if h <= BadDllLoad then
      ShowMessage('Bad Dll Load')
   else begin
      Bmp := TBitmap.Create;
      //获得所定义的位图
      Bmp.Handle := LoadBitmap(H,'BITMAP_1');
      Canvas.Draw(0, 0, Bmp);
      Bmp.Free;
      FreeLibrary(h);
   end;
end;
```

第七章

文件、目录及驱动器

热点透视

Illustrator 中的绘图是指用颜色、渐变和图案填充对象，并将颜色和图案应用到对象的边框之上。本章详细介绍了颜色、渐变与图案的使用。用户可以产生和编辑颜色，并用来绘图。同时还可以命名和保存颜色，建立自定义的调色板。渐变是 Illustrator 的一个特殊功能，Illustrator 提供对渐变视觉效果的许多控制。用户可以自己创建效果惊人的彩色渐变，并用其填充对象。图案在 Illustrator 中用于对象的填充、边框的重复和装饰性的设计。图案给用户提供了许多的创意和功能。本章还介绍了 Illustrator 的图层功能，利用图层用户可以将图画组织在不同的层上，并将其作为各自独立的单元来编辑和浏览。



获得可用驱动器及其类型



遇到难题

应用程序往往需要知道本台计算机中到底安装了哪些驱动器，以及这些驱动器的类型。例如，如果应用程序为杀毒软件，并且只希望在本机使用，那么应用程序中应该判断所指定的驱动器是本地的还是远程的，如果是远程驱动器（网络驱动器）的话，杀毒软件应该提示本程序不支持这项功能。所以应用程序除了需要了解本机安装了多少个驱动器外，还需要知道这些驱动器是什么样的类型，即是软驱、光驱、硬盘，还是网络驱动器等。



钥匙在此

调用 Win32 API 函数 GetDriveType（）可以获得驱动器列表。该函数的定义格式为：

```
function GetDriveType(lpRootPathName: Pchar): UINT;
```

这个函数需要传递一个 Pchar 类型的参数，该参数为驱动器的根目录，如“C:\”。该函数返回一个整数值指定驱动器的类型，如表 7-1 所示。

表 7-1　　GetRriveType（）返回值

返回值	含　义
0	不能确定驱动器类型
1	根目录不存在
DRIVE_REMOVABLE	可移动驱动器
DRIVE_FIXED	不可移动驱动器
DRIVE_REMOTE	远程（网络）驱动器
DRIVE_CDROM	CD-ROM 驱动器
DRIVE_RAMDISK	RAM 驱动器

下面的例子将获取可用的驱动器名，并用不同的图标来表示不同的驱动器类型。



跟我来

新建应用程序，然后在设计窗体上加上一个按钮、一个 TiamgeList 和一个 TlistView 部件，其实例分别为 btnGetDriveTypes、DriveImageList 和 DriveListView。

为 DriveImageList 添加图标，分别代表不同的驱动器类型。然后设置 DriveListView 的 LargeImageList 属性值为 DriveImageList，并为按钮的单击事件添加以下代码：

清单 7-1　获得可用驱动器及其类型

```
procedure TmainForm.btnGetDriveTypesClick(Sender: Tobject);
var
   I: Integer;
   C: String;
   Dtype: Integer;
   DriveString: String;
   DriveImageIndex:integer;
   TmpItem:TlistItem;
begin
   { 从 A..Z 中确定哪些是可使用的驱动器}
   for I := 65 to 90 do
   begin
      // 生成代表驱动器的根目录字符串
      C := chr(I)+':\';
      { 调用 GetDriveType（） 函数返回一个代表驱动器类型的整数值}
      Dtype := GetDriveType(Pchar©);
      { 根据返回值，在 TlistView 上添加相应的驱动器名称以及图标 }
      case Dtype of
         0: DriveString := C+'驱动器类型无法确定';
         1: DriveString := C+'根目录不存在';
         DRIVE_REMOVABLE:
            begin
               DriveString := C+'为可移动驱动器';
               DriveImageIndex:=0;
            end;
      DRIVE_FIXED:
         begin
            DriveString := C+'为不可移动驱动器';
            DriveImageIndex:=1;
         end;
      DRIVE_REMOTE:
         begin
            DriveString := C+'为网络驱动器';
            DriveImageIndex:=3;
         end;
```

```
        DRIVE_CDROM:
          begin
              DriveString := C+’ 为 CD-ROM 驱动器’;
              DriveImageIndex:=2;
          end;
        DRIVE_RAMDISK:
          begin
              DriveString := C+’ 为 RAM 磁盘.’;
              DriveImageIndex:=4;
          end;
    end;
    // 仅仅添加有效的驱动器
        if not ((Dtype = 0) or (Dtype = 1)) then begin
          TmpItem:=DriveListView.Items.Add;
          TmpItem.Caption:=C;
          TmpItem.ImageIndex:=DriveImageIndex;
        end;
      end;

  end;
```

清单 7-1 中以 A..Z26 个字符为序进行循环，以判断它们中有哪些是合法的驱动器。对于合法的驱动器，GetDriveType（）将返回驱动器的类型，Case 语句根据返回的类型值合成一个描述性的字符串，并且指定所使用图标的索引号。当返回的驱动器为可用时，则将其加入到 DriveListView 中，并将其显示出来。

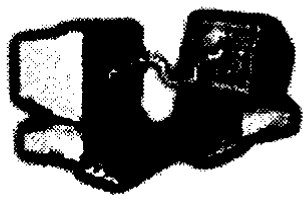
运行该程序代码后单击按钮便可以获得可用驱动器及其类型，运行结果如图 7-1 所示。

图 7-1　获得可用驱动器及其类型



再送礼包

Delphi 自带的 TdriveComboBox 控件可以用来选择一个驱动器，它可以在控件面板的 Win31 页上找到。



获取驱动器信息



遇到难题

当获得了可用的驱动器及其类型之后，就可以获取某个驱动器的信息了。驱动器具有其自身的属性，当使用资源管理器查看驱动器的属性时，则在属性的“常规”页面中将包括如下的信息：如卷标、类型、文件系统、已用空间、可用空间以及容量等。如何在应用程序中使用这些驱动器的信息呢？



钥匙在此

要获得上述驱动器的信息，需要调用 Win32 API 函数。因为要获得驱动器的属性信息，不是靠一个函数就可以实现的，所以下面分别进行描述。

1. 检测驱动器是否准备好

要获得某个驱动器的信息，首先应当保证该驱动器是可以访问的，也就是说是准备就绪的。如，对于软驱应当保证软驱中有软盘，对于光驱要保证光驱中带有光盘，这样，才能访问该驱动器的一些属性。

检测驱动器是否准备就绪，并无直接的函数可以利用。通常可以采取以下的方式实现这一目的，先改变目录为指定的驱动器符号，如果出错，则表示当前指定的驱动器并未准备好。此时系统会自动提示“未就绪”等错误信息。但是为了使自己的程序能控制该过程，有时需要使 Windows 的系统提示不弹出来，而由自己对应用程序进行控制。为了达到这一目的，需要调用 Win32 API 函数 SetErrorMode（），其定义格式如下：

function SetErrorMode(uMode: UINT): UINT;

其参数为指定系统对错误进行处理的方式的标志位。例如，当该参数设置为

SEM_NOOPENFILEERRORBOX 时，则表示系统禁止弹出错误提示对话框。对于其他错误处理方式的标志位常量请参见该函数的帮助文件。清单 7-2 为一个检测驱动器是否准备

就绪的例程。

清单 7-2　检测驱动器是否就绪

```
function IsDriveReady(DriveLetter : char) : bool;
var
   OldErrorMode : Word;
   OldDirectory : string;
begin
   //禁止系统弹出错误提示对话框
   OldErrorMode := SetErrorMode(SEM_NOOPENFILEERRORBOX);
   GetDir(0, OldDirectory);
   {$I-} //关闭 I/O 函数的错误检测，防止异常发生
      ChDir(DriveLetter + ':\');
   {$I+}
   //IoResult 返回上一次 I/O 操作的错误代码，若返回值为 0 则表明并未发生错误
    if IoResult <> 0 then
      Result := False
    else
      Result := True;

   //恢复原有设置
   ChDir(OldDirectory);
   SetErrorMode(OldErrorMode);
end;
```

2. 获取驱动器的大小和可用空间信息

驱动器的大小信息主要包括：

- 每一簇的扇区数
- 每一扇区的字节数
- 可用簇数
- 簇的总数
- 可用磁盘字节数
- 磁盘大小

调用 Win32 API 函数 GetDiskFree（）可以得到上述第 1 至第 4 项信息。当然第 5 项和第 6 项可以通过对前面获得的信息进行计算而获得，也可以通过调用 Delphi 的 sysUtils.pas 单元中声明的 DiskFree（）和 DiskSize（）函数来获取。

GetDiskFree（）函数的定义文法如下：

```
function GetDiskFreeSpace(lpRootPathName: Pchar;
   var      lpSectorsPerCluster,      lpBytesPerSector,      lpNumberOfFreeClusters,
```

lpTotalNumberOfClusters: DWORD): BOOL;

其中，参数 lpRootPathName 表示所指定驱动器的根目录，如“c:\”。其他参数都为变量参数，调用该函数后，系统将把相应的信息存贮在这些参数中，以供调用程序使用。LpSectorsPerCluster 表示每簇的扇区数，lpBytesPerSector 表示每个扇区的字节数，lpNumberOfFreeClusters 表示可用簇数，lpTotalNumberOfClusters 表示总共的簇数。

3. 获取有关卷的信息

驱动器中的卷标、文件系统以及序号等都属于驱动器的卷信息。要获取驱动器的卷信息，可以通过调用系统函数 GetVolumeInformation（）来实现。该函数的定义文法如下：

```
function GetVolumeInformation(lpRootPathName: Pchar;
  lpVolumeNameBuffer: Pchar; nVolumeNameSize: DWORD; lpVolumeSerialNumber:
PDWORD;
    var lpMaximumComponentLength, lpFileSystemFlags: DWORD;
      lpFileSystemNameBuffer: Pchar; nFileSystemNameSize: DWORD): BOOL;
```

其中，参数 lpRootPathName 表示指定驱动器的根目录，参数 pVolumeNameBuffer 和 nVolumeNameSize 分别存放卷标信息的内容和大小，LpVolumeSerialNumber 表示卷的序列号，其余参数都是变量参数，用于存储系统返回的有关信息。NfileSystemNameSize 表示存储文件系统的类型，如 FAT、FAT32、NTFS 以及 CDFS 等。

下面看一个完整的例子。



跟我来

清单 7-3 为清单 7-1 的加强版，在原有的基础上进行了改进。本例中除了能获得可用驱动器及其类型外，还可以获得某一准备就绪的驱动器的有关信息，如驱动器的容量、可用空间以及卷的相关信息等。

清单 7-3　获取驱动器信息

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, FileCtrl, ComCtrls, ImgList;

type

  TmainForm = class(Tform)
    btnGetDriveTypes: Tbutton;
```

```
    DriveListView: TlistView;
    DriveImageList: TimageList;
    DrvInfoGroupBox: TgroupBox;
    lblSectPerClust2: Tlabel;
    lblBytesPerSector2: Tlabel;
    lblNumFreeClusters2: Tlabel;
    lblTotalClusters2: Tlabel;
    lblSectPerCluster: Tlabel;
    lblBytesPerSector: Tlabel;
    lblNumFreeClust: Tlabel;
    lblTotalClusters: Tlabel;
    lblFreeSpace2: Tlabel;
    lblTotalDiskSpace2: Tlabel;
    lblFreeSpace: Tlabel;
    lblTotalDiskSpace: Tlabel;
    Label1: Tlabel;
    lblVolumnName: Tlabel;
    lblVolumeSerialNo: Tlabel;
    lblMaxComponentLength: Tlabel;
    lblFileSystemFlags: Tlabel;
    lblFileSystemName: Tlabel;
    Label2: Tlabel;
    Label3: Tlabel;
    Label4: Tlabel;
    Label5: Tlabel;
    procedure btnGetDriveTypesClick(Sender: Tobject);
    procedure DriveListViewClick(Sender: Tobject);
  end;

var
  MainForm: TmainForm;

implementation

{$R *.DFM}
const
    MaxPath=256;

function IsDriveReady(DriveLetter : char) : bool;
var
  OldErrorMode : Word;
  OldDirectory : string;
begin
  //禁止系统弹出错误提示对话框
  OldErrorMode := SetErrorMode(SEM_NOOPENFILEERRORBOX);
  GetDir(0, OldDirectory);
```

```
    {$I-} //关闭 I/O 函数的错误检测，防止异常发生
      ChDir(DriveLetter + ':\');
    {$I+}
    //IoResult 返回上一次 I/O 操作的错误代码，若返回值为 0 则表明并未发生错误
     if IoResult <> 0 then
      Result := False
     else
      Result := True;

    //恢复原有设置
    ChDir(OldDirectory);
    SetErrorMode(OldErrorMode);
end;
procedure TmainForm.btnGetDriveTypesClick(Sender: Tobject);
var
   I: Integer;
   C: String;
   DriveString: String;
   Dtype: Integer;
   DriveImageIndex:integer;
   TmpItem:TlistItem;
begin
   { 从 A..Z 循环确定可利用的驱动器}
   for I := 65 to 90 do
   begin
      // 生成代表驱动器的根目录的字符串
      C := chr(I)+':\';
      { 调用 GetDriveType（） 函数以返回一个代表驱动器类型的整数值}
      Dtype := GetDriveType(Pchar©);
      { 基于返回值，在 TlistView 上添加相应的驱动器名称以及图标 }
      case Dtype of
         0: DriveString := C+' 驱动器类型无法确定';
1: DriveString := C+' 根目录不存在';
DRIVE_REMOVABLE:
   begin
       DriveString := C+' 为可移动驱动器';
       DriveImageIndex:=0;
   end;
DRIVE_FIXED:
   begin
       DriveString := C+' 为不可移动驱动器';
       DriveImageIndex:=1;
   end;
DRIVE_REMOTE:
   begin
```

```
                DriveString := C+' 为网络驱动器';
                DriveImageIndex:=3;
            end;
        DRIVE_CDROM:
            begin
                DriveString := C+' 为 CD-ROM 驱动器';
                DriveImageIndex:=2;
            end;
        DRIVE_RAMDISK:
            begin
                DriveString := C+' 为 RAM 磁盘.';
                DriveImageIndex:=4;
            end;
    end;
    // 仅添加有效的驱动器
    if not ((Dtype = 0) or (Dtype = 1)) then begin
        TmpItem:=DriveListView.Items.Add;
        TmpItem.Caption:=C;
        TmpItem.ImageIndex:=DriveImageIndex;
        TmpItem.Data:=Pointer(I);
    end;
  end;

end;

procedure TmainForm.DriveListViewClick(Sender: Tobject);
var
  RootPath: String;                 // Holds the drive root path
  SectorsPerCluster: Dword;    // Sectors per cluster
  BytesPerSector: Dword;         // Bytes per sector
  NumFreeClusters: Dword;       // Number of free clusters
  TotalClusters: Dword;          // Total clusters
  FreeSpace: Int64;                  // Free space on drive
  TotalSpace: Int64;                // Total drive space.
  DriveByte: Byte;                    // Drive byte value
  DriveNum:   Integer;                // Drive number 1 = A, 2 = B, etc.

  VolumeName,
  FileSystemName        : array [0..MAX_PATH-1] of Char;
  VolumeSerialNo        : Dword;
  MaxComponentLength,
  FileSystemFlags       : Dword;
begin
  with DriveListView do
  begin
    if SelCount<=0 then exit;
```

```
    { 将代表驱动器字母的 ASCII 值转换为有效的驱动器号（整数）:
      1 = A, 2 = B, 等}
DriveByte := Integer(selected.data)-64;
{ 从所选择的项的标题中获得该驱动器的根目录}
RootPath := char(selected.data)+':\';
//如果驱动器已准备好，则继续执行
if IsDriveReady(char(selected.data)) then
begin
   { 调用 GetDiskFreeSpace（）获取驱动器信息}
   if GetDiskFreeSpace(Pchar(RootPath), SectorsPerCluster,
    BytesPerSector, NumFreeClusters, TotalClusters) then
   begin
     {如果该函数执行成功，则修改显示磁盘信息的标签}
     DrvInfoGroupBox.Caption:='当前驱动器'+Selected.Caption;
     lblSectPerCluster.Caption := Format('%.0n', [SectorsPerCluster*1.0]);
     lblBytesPerSector.Caption := Format('%.0n', [BytesPerSector*1.0]);
     lblNumFreeClust.Caption := Format('%.0n', [NumFreeClusters*1.0]);
     lblTotalClusters.Caption := Format('%.0n', [TotalClusters*1.0]);
     // 获得可用的磁盘空间
     FreeSpace   := DiskFree(DriveByte);
     TotalSpace := DiskSize(DriveByte);
     lblFreeSpace.Caption := Format('%.0n', [FreeSpace*1.0]);
     { 计算这个磁盘空间 }
     lblTotalDiskSpace.Caption := Format('%.0n', [TotalSpace*1.0]);

     //获取驱动器的卷信息
     GetVolumeInformation(Pchar(Rootpath),VolumeName,MAX_PATH,@VolumeSeria
lNo,
                          MaxComponentLength,FileSystemFlags,
                          FileSystemName,MAX_PATH);
     lblVolumnName.Caption:=VolumeName;      //卷标
     lblVolumeSerialNo.Caption:='$'+IntToHex(VolumeSerialNo,8);
     lblMaxComponentLength.Caption:=IntToStr(MaxComponentLength);
     lblFileSystemFlags.Caption:='$'+IntToHex(FileSystemFlags,4);
     lblFileSystemName.caption:=FileSystemName; //文件系统

   end;
  end
  else          ShowMessage('磁盘未准备好！');
 end;

end;

end.
```

在该清单中，如果单击 bt¬GetDriveTypes 按钮，程序就会将可用驱动器及其类型信息在 DriveListView 上通过图标显示出来，并将驱动器号（整数）存入各 TlistItem 的 data 数据中。当单击 DriveListView 的驱动器的图标时，首先验证该驱动器是否准备就绪，如果未准备好则程序给出提示，否则将读出该驱动器的有关信息，并将这些信息内容显示到相应的标签中。运行该程序，结果如图 7-2 所示。

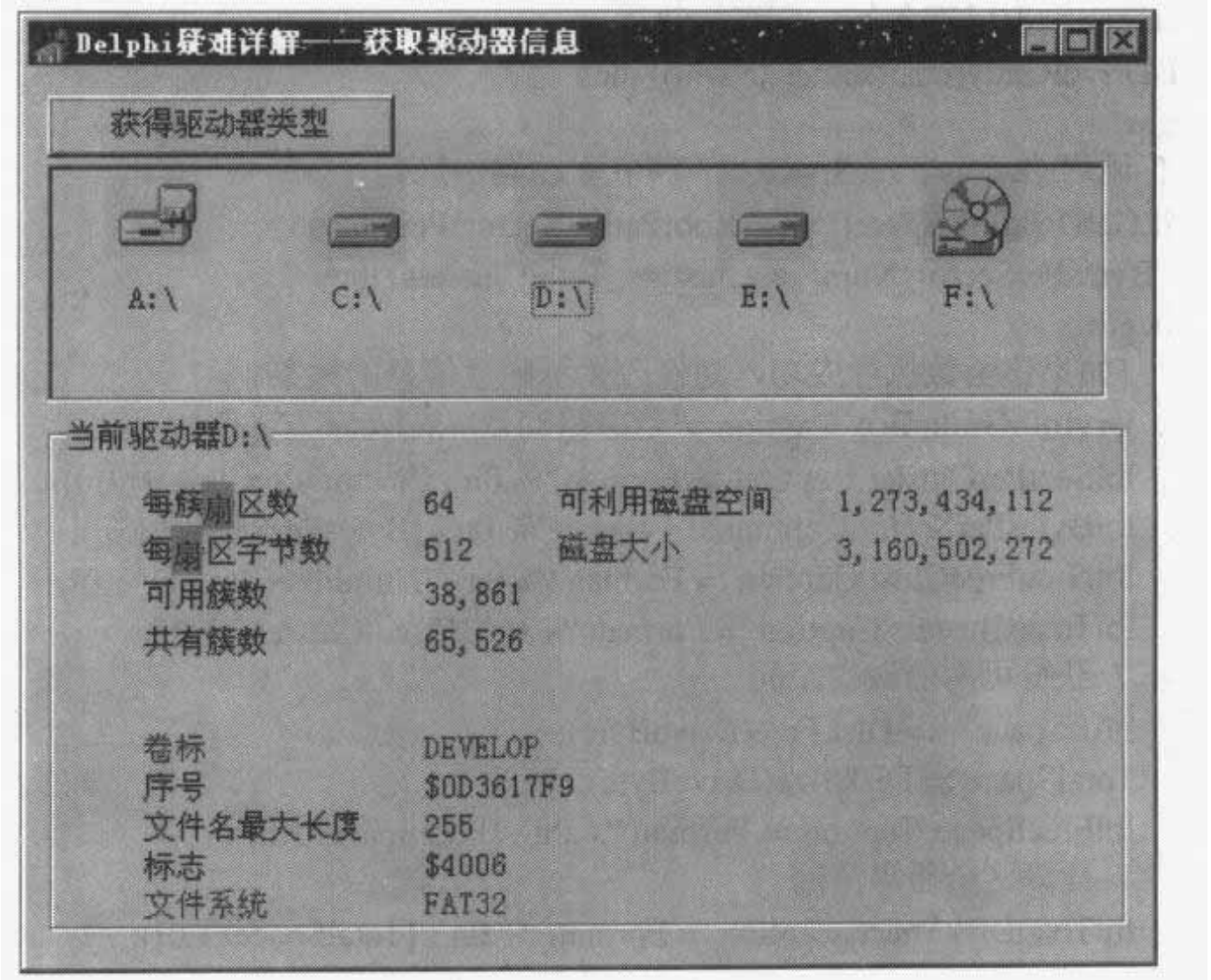


图 7-2　获取驱动器信息



获取几种常用目录



遇到难题

在程序中经常需要知道自己当前所在的目录，有时在安装中还需要知道 Windows 及其系统目录在什么地方等。又如，程序在进行数据处理时，还要进行数据的临时交换，并将自己的中间数据放在系统的临时目录中。如果应用程序被放入了 Windows 的安装盘中，则在重新安装应用程序的时候，还需要知道 Windows 的安装路径。这些目录其实都可以在应用程序中获取。



钥匙在此

下面对几种常用目录的获取分别进行介绍。

1. 获取 Windows 目录

为了获取 Windows 目录，需要调用 Win32 API 函数 GetWindowsDirectory（）。这个函数的声明如下：

```
function GetWindowsDirectory(lpBuffer: Pchar; uSize: UINT): UINT;
```

其中，第一个参数用于返回 Windows 的目录，为 Pchar 类型的字符串，第二个参数用于指定缓冲区的大小。下面的例子显示了这个函数的用法。

```
{$IFNDEF WIN32}
 const MAX_PATH = 144;
{$ENDIF}

procedure Tform1.Button1Click(Sender: Tobject);
var
   a : Array[0..MAX_PATH] of char;
begin
   GetWindowsDirectory(a, sizeof(a));
   ShowMessage(StrPas(a));
end;
```

上述代码中的 a 定义为数组字符串，由于 showmessage（）的参数为 string 类型，因此，在需要将 null 结尾的数组字符串转换为 Pascal 字符串时应调用 strPas（）函数。另外，将 Pascal 字符串转换为以 null 结尾的字符串时，需要调用 StrPCopy（）函数。

2. 获取系统目录

为了获取系统目录（该目录中含有很多重要的动态连接库），需要调用 Win32 API 函数 GetSystemDirectory（）。该函数的使用方法和 GetWindowsDirectory（）函数类似，不同的是，它返回的是系统 system 目录的完整路径，而不是 Windows 目录。下面的代码显示了这个函数的用法。

```
{$IFNDEF WIN32}
 const MAX_PATH = 144;
{$ENDIF}

procedure Tform1.Button1Click(Sender: Tobject);
var
   a : Array[0..MAX_PATH] of char;
begin
```

```
    GetSystemDirectory(a, sizeof(a));
    ShowMessage(StrPas(a));
end;
```

3. 获取系统临时目录

为了获取临时目录，需要调用 Win32 API 函数 GetTempPath（）。其使用方法与上述两个函数的使用方法类似，只不过参数的顺序不一致，而且返回的是完整的临时目录路径。其声明格式如下：

```
function GetTempPath(nBufferLength: DWORD; lpBuffer: Pchar): DWORD;
```

下面的代码显示了该函数的用法：

```
{$IFNDEF WIN32}
 const MAX_PATH = 144;
{$ENDIF}

procedure Tform1.Button1Click(Sender: Tobject);
var
   a : Array[0..MAX_PATH] of char;
begin
   GetTempPath(sizeof(a),a);
   ShowMessage(StrPas(a));
end;
```



如果环境变量已经定义了 TMP 或者 TEMP 的话，则该函数将取得环境变量所定义的临时路径，否则以 Windows 目录下的 TEMP 为临时路径。

4. 获取当前目录

在应用中，经常需要知道应用程序本身所在的位置，即应用程序执行的目录。这可以通过调用 Win32 API 函数 GetCurrentDirectory（）来获取。该函数与 GetTempPath（）函数的用法完全类似，只是它同时还获取应用程序当前的位置路径。

```
{$IFNDEF WIN32}
 const MAX_PATH = 144;
{$ENDIF}

procedure Tform1.Button1Click(Sender: Tobject);
var
   a : Array[0..MAX_PATH] of char;
begin
   GetCurrentDirectiory(sizeof(a),a);
   ShowMessage(StrPas(a));
end;
```

5. 获取 Windows 的安装路径

有时需要从 Windows 的安装盘中安装一些文件。Windows 的安装文件都在 CAB 文件中，安装时有的是从 CD-ROM 中进行的，有的是从用户的硬盘中进行的。如何找到 Windows 是在哪儿安装的呢？这可以通过 Windows 的注册表来读取，Windows 的原始安装目录注册在“HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\SETUP 的 ”SourcePath 键值中。

以下代码可以获取 Windows 的原始安装路径。

```
procedure Tform1.Button1Click(Sender: Tobject);
var
  reg: Tregistry;
begin
  reg := Tregistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\SETUP',
              false);
  ShowMessage(reg.ReadString('SourcePath'));
  reg.CloseKey;
  reg.free;
end;
```



跟我来

清单 7-4 是综合了上述内容的完整例子。注意在 uses 语句中加入 register 单元。

清单 7-4　获取常用目录

```
procedure TfrmMain.Button1Click(Sender: Tobject);
var
  a : Array[0..MAX_PATH] of char;
  reg: Tregistry;
begin
  GetwindowsDirectory(a, sizeof(a));
  lblwindows.Caption:=strpas(a);
  GetSystemDirectory(a, sizeof(a));
  lblSystem.caption:=StrPas(a);
  GetTempPath(sizeof(a),a);
  lblTmp.Caption:=strpas(a);
  GetCurrentDirectory(sizeof(a),a);
```

```
    lblCurrentDir.Caption:=strpas(a);

    //获取 Windows 的原始安装路径
    reg := Tregistry.Create;
    reg.RootKey := HKEY_LOCAL_MACHINE;
    reg.OpenKey('Software\Microsoft\Windows\CurrentVersion\SETUP',
                false);
    lblSetup.caption:=reg.ReadString('SourcePath');
    reg.CloseKey;
    reg.free;
end;
```

该程序首先调用 Win32 API 函数获取几个常用目录，接着读取注册表中的信息来获取 Windows 的原始安装目录。运行该程序的结果如图 7-3 所示。

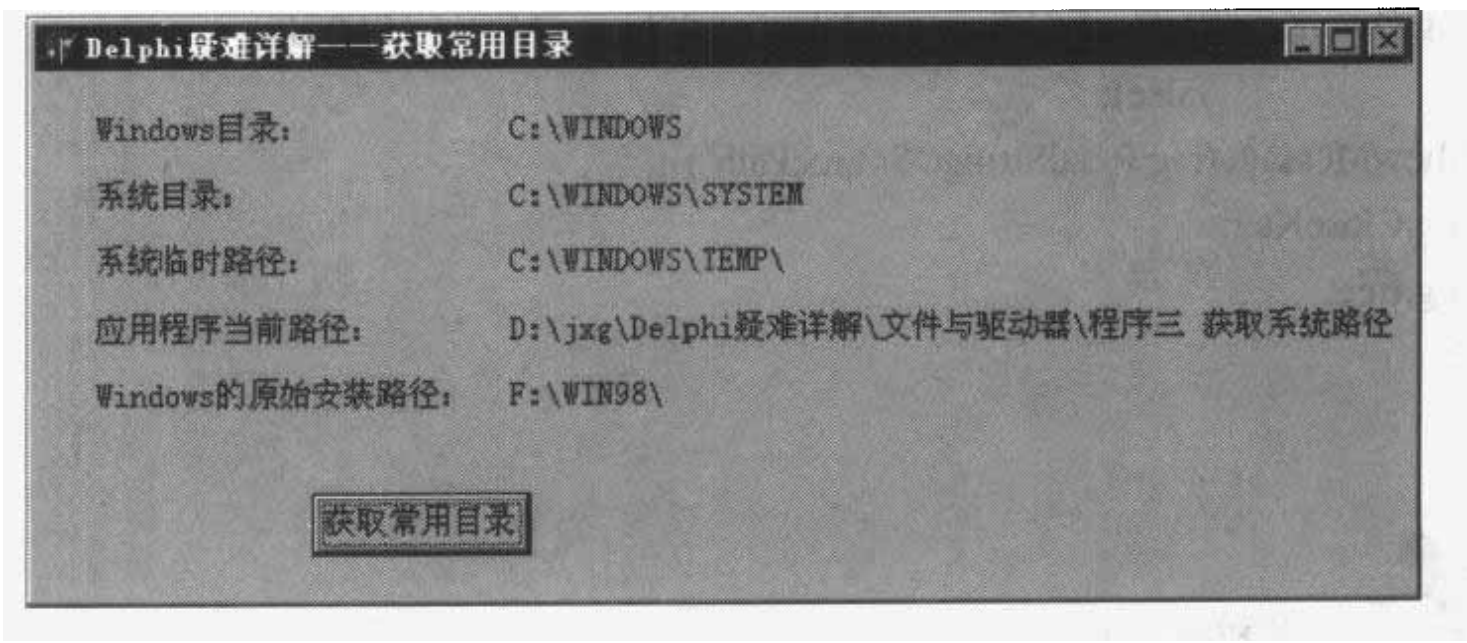


图 7-3　获取常用目录



再送礼包

为了获取应用程序的当前路径，还可以直接使用 Delphi 中的 System.pas 单元中的 CurDir（）函数或者 SysUtils.pas 单元中声明的 GetCurrentDir（）函数。

另外，Delphi 提供了很多处理目录和文件名的函数。例如，从 Tapplication.ExeName 特性中就可以获取当前应用程序的完整路径和文件名。假设一个可执行文件的路径为 c:\temp\a.exe，表 7-2 列出了将 Tapplication.ExeName 特性传递给一些函数的返回值。

表 7-2　　文件名/目录处理函数（假设参数值为 c:\temp\a.exe）

函　数	描 述	结 果
ExtractFileName（）	获取文件名	a.dat
ExpandFileName（）	通过插入当前驱动器及目录来返回整个文件及路径名。例如，则当前目录为 c:\temp, 如果 filename=a.dat->c:\temp\a.dat	c:\temp\a.dat

续表

函　数	描 述	结 果
ExtractFilePath（）	返回 Filename 中的路径名	c:\temp\
ExtractFileDir（）	返回 Filename 中文件的所在目录	c:\temp
ExtractFileExt（）	返回文件的后缀名	.dat
ExtractFileDrive（）	返回 Filename 中的驱动器符号	c:



查找文件



遇到难题

在程序对一些文件进行操作时，如果要处理的文件名不是固定的几个文件，或者是需要用户进行交互地选择一些文件的话，则常常需要应用程序能支持在指定目录下查找某种格式文件的功能，而不是仅查找单个文件。此时，就要求在查找文件时能支持通配符，并能在指定目录的子目录下进行查询。



钥匙在此

查找文件主要涉及到三个函数和一个数据结构。三个函数为 FindFirst（）、FindNext（）、FindClose（），一个数据结构为 TsearchRec。

对于实现文件的查找，一般按照以下几个步骤：

（1）使用 FindFirst（）获取与指定条件相匹配的第一个文件；

（2）使用 FindNext（）函数获取与 FindFirst（）指定条件相匹配的下一个文件；

（3）最后调用 FindClose（）释放 FindFirst（）所分配的内存。

下面将对这三个函数进行解释。

1. FindFirst（）函数

声明格式：function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer;

功能描述：打开一个查找句柄，并将符合查找条件的第一个文档信息存入变量 F 中。如果找到符合条件的文件信息时将返回 0，否则将返回一个错误的 Windows 代码。

参数：path 为所要查找文件的路径加上要查找文件的 mask，如 c:\temp*.*。Attr 为要查找文件的属性，例如 faSysFile　表示系统文件，faDirectory 表示目录文件，该属性可以组合，如 faSysFile+faDirectory。F 为符合该查找条件的文件的信息。对于该函数而言，F 表示符合该查找条件的第一个文件的信息。F 参数包含了以下信息：

TSearchRec = record

```
Time: Integer;  //文件的时间信息
Size: Integer;//文件的大小
Attr: Integer;//文件的属性
Name: TFileName;//DOS 文件名
ExcludeAttr: Integer;//文件属性
FindHandle: THandle;//查找的文件句柄，该值是查找下一个文件的纽带
FindData: TWin32FindData;//文件的附加信息数据，具体参见 Win32 SDK.
```

2. FindNext（）函数

声明格式：function FindNext(var F: TSearchRec): Integer;

功能描述：查找与 FindFirst（）查找条件相匹配的下一个文件，并将该文件信息存入变量 F 中。如果查找成功，则返回 0。

3. Findclose（）函数

声明格式：procedure FindClose(var F: TSearchRec);

功能描述：释放 FindFirst（）函数开辟的空间。调用该函数将结束 FindFist/findNext 的操作序列。

下面举一个调用这三个函数的例子：

```
var
  sr: TSearchRec;
  FileAttrs: Integer;
begin
    if FindFirst('c:\temp\*.*', FileAttrs, sr) = 0 then   begin
        showmessage(sr.Name);
        while FindNext(sr) = 0 do begin
            showmessage(sr.name);        findfirstfile
        end;
        findclose(sr);
     end;
end;
```

该过程将报告 c:\temp 目录下的所有文件的文件名。



跟我来

有时，需要在一个目录中查找某种格式的文件。清单 7-4 显示了如何通过递归调用来实现对当前目录和子目录的搜索。

清单 7-4　在目录中查找文件

```
unit MainFrm;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, FileCtrl, Grids, Outline, DirOutln, ComCtrls;

type
  TMainForm = class(TForm)
    dcbDrives: TDriveComboBox;
    edtFileMask: TEdit;
    lblFileMask: TLabel;
    btnSearchForFiles: TButton;
    lbFiles: TListBox;
    dolDirectories: TDirectoryOutline;
    CheckBox1: TCheckBox;
    procedure btnSearchForFilesClick(Sender: TObject);
    procedure dcbDrivesChange(Sender: TObject);
  private
    FFileName: String;
    function GetDirectoryName(Dir: String): String;
    procedure FindFiles(APath: String);
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}

function TMainForm.GetDirectoryName(Dir: String): String;
{使目录成为有效的目录名 }
begin
  if Dir[Length(Dir)]<> '\' then
    Result := Dir+'\'
  else
    Result := Dir;
end;

procedure TMainForm.FindFiles(APath: String);
{ 该函数为递归函数，以便可以在当前目录以及子目录下查找文件 }
```

```
var
  FSearchRec,
  DSearchRec: TSearchRec;
  FindResult: integer;

  function IsDirNotation(ADirName: String): Boolean;
  begin
    Result := (ADirName = '.') or (ADirName = '..');
  end;

begin
  APath := GetDirectoryName(APath); //  获取有效的目录名
{  找到与指定文件相匹配的第一个文件名  }
FindResult := FindFirst(APath+FFileName,faAnyFile+faHidden+
                                faSysFile+faReadOnly,FSearchRec);
try
  {  根据指定的格式，继续进行查找，并将查找结果加入 listbox 中}
  while FindResult = 0 do
  begin
    lbFiles.Items.Add(LowerCase(APath+FSearchRec.Name));
    FindResult := FindNext(FSearchRec);
  end;

  {  根据“是否包含子目录进行查找”选项，决定是否在子目录中查找。如果
  将查找当前目录的子目录下的文件，则是一个递归调用  }
  if checkbox1.checked then begin
      FindResult := FindFirst(APath+'*.*', faDirectory, DSearchRec);

      while FindResult = 0 do
      begin
        if ((DSearchRec.Attr and faDirectory) = faDirectory) and not
            IsDirNotation(DSearchRec.Name) then
            FindFiles(APath+DSearchRec.Name); //  此处为递归调用
            FindResult := FindNext(DSearchRec);
        end;
    end;
  finally
    FindClose(FSearchRec);
  end;
end;
```

```
procedure TMainForm.btnSearchForFilesClick(Sender: TObject);
{ 该方法开始进行查找过程。由于查找文件可能需要花一定的时间，于是首先将鼠标指针变成漏斗状，然后清空列表框。接着，递归调用 FindFiles（）函数在子目录中进行查找}
begin
  Screen.Cursor := crHourGlass;
  try
    lbFiles.Items.Clear;
    FFileName := edtFileMask.Text;
    FindFiles(dolDirectories.Directory);
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TMainForm.dcbDrivesChange(Sender: TObject);
begin
  dolDirectories.Drive := dcbDrives.Drive;
end;

end.
```

在 FindFiles（）函数中，先在当前的目录中查找文件，然后根据是否包含子目录进行查找的选项来决定是否进行递归调用以实现在子目录中的查找。

图 7-4 为该程序在指定路径下查找*.pas 文件的执行结果。

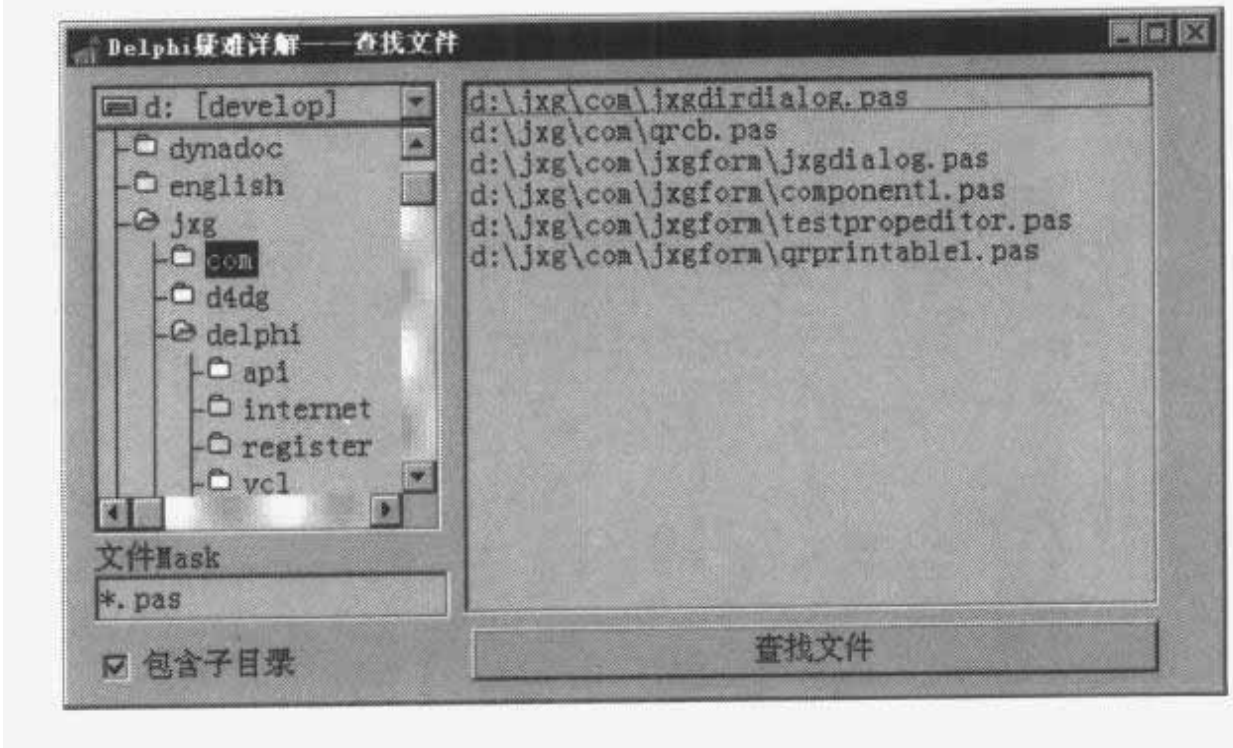


图 7-4　查找文件



再送礼包

上述三个函数为 Delphi5 对系统函数的一个包装。对应的 Win32 API 中的原型函数分别为 FindFirstFile（）、FindNextFile（）和 FindCloseFile（）函数，具体参数含义可参见相应的帮助文件。

第八章

剪贴板

热点透视

对于 Windows 程序员来说，剪贴板是再熟悉不过了。DOS 下多个应用程序共享信息用的是存储文件，而到了 Windows 时代，剪贴板则是多个程序之间共享数据信息最简单的方式之一。因此如何灵活的应用剪贴板技术是 Windows 程序员应具备的技术。几乎所有带“编辑”菜单的应用程序都使用剪贴板。究竟什么是剪贴板呢？简单的说，剪贴板就是一块内存区域和 Win32 API 函数集，通过这些函数，应用程序能够从内存区域中保存和获取信息。例如，可以从 Delphi 的代码编辑器复制部分源代码到 Windows 的记事本或其他文本编辑器中。

本章将详细介绍剪贴板技术。如何在程序之间通过剪贴板实现数据的复制和粘贴，应用程序如何能够自动感知其他程序对数据的复制或剪贴，程序员又是如何定义自己的剪贴板格式，这些都是本章要讲述的内容。



复制、粘贴文本和图形数据



遇到难题

在 Windows 中我们已经习惯了使用剪贴板来复制数据。例如，Windows 中的“记事本”可以将一些文本信息复制到剪贴板中；“画笔”可以将选择的一些图形区域复制到剪贴板中；微软的 IE 浏览器 （Internet Explorer）可以复制 JPEG 格式文件到剪贴板中；在 Delphi 的开发环境中可以复制控件到剪贴板中并且又可以将这些控件进行粘贴。

这些都是从使用者的角度来看问题的。但对于 Windows 程序员来讲，要考虑的是如何使程序也支持这种剪贴板技术，即能够从应用程序中将数据拷贝至剪贴板中供其他应用程序使用，以及从其他应用程序中拷贝数据至剪贴板以供正被编写的程序使用。另外，在“画笔”中拷贝的数据在“记事本”中为何粘贴不了。有人说，记事本中存放的是文本数据，而画笔拷贝的则是图形数据。其实这看到的只是表面现象，剪贴板中究竟放的是什么数据，如何将其取出来，这些都是本节要解决的问题。



钥匙在此

1. 认识 Delphi 中的剪贴板

在学习 Delphi 之前，您不得不直接调用各种各样的剪贴板函数来实现复制和粘贴，而且还要保证应用程序不受错误的剪贴板内容的影响。有了 Delphi 后，只要使用一个叫 Clipboard 的全局变量即可实现。Clipboard 是一个对象，它封装了 Win32 的剪贴板。

Windows 的剪贴板（Clipbord）就如同一个数据包，载有文本、图形以及其他信息，这些信息被应用程序用来进行剪切、拷贝和粘贴。TClipBoard 是 Windows 剪贴板被 Delphi 封装的一个类，这个类大大减少了 Delphi 程序员的工作量。

TClipBoard 的属性用于提供剪贴板的格式信息。

TClipBoard 的方法主要用于：

（1）根据格式获取剪贴板中的数据；

（2）打开和关闭剪贴板；

（3）管理和操作剪贴板中的对象格式。



在使用该类时注意：

（1） 不必调用 constructor 来创建单独的 TClipboard 对象，相反，可以通过调用剪

贴板的函数来使用全局的剪贴板实例。这样，当允许应用程序共享 TClipBoard 单一实例时，通过使用全局的实例可以安全维护而不会删除剪贴板对象。

（2） 确保使用 uses 语句包含 clipbrd 单元。

2. 剪贴板格式

为什么 Win32 需要函数集和消息来操纵剪贴板呢？向剪贴板复制信息，不仅仅是分配一块内存区域并将数据放到该区域中，其他应用程序还需要知道如何去得到数据，以及数据是什么样的格式。剪贴板中数据的格式可由以下语句获得：

TClipboard.HasFormat

Win32 目前支持 25 种预定义的格式。常用的格式见表 8-1。

表 8-1　　Format 参数常用值

值	含义
CF_TEXT	表示每行带有 CR-LF 结尾的文本，文本末以一个空字符结尾
CF_BITMAP	表示是位图格式
CF_METAFILEPICT	表示图形文件格式
CF_PICTURE	表示一种 Tpicture 对象
CF_OBJECT	表示任何 persistent 对象

作为对象，Clipboard 应有检测剪贴板格式的方法，该函数为 HasFormat（），现说明如下：

定义格式：Function HasFormat（Format: Word）: Boolean;

描述：通过使用该函数可以知道在剪贴板中的信息是否使用了指定格式进行编码。如果返回值为 True，则表明 Format 为当前格式；如果返回值为 False，则表明剪贴板中的格式不是 Format 指定的格式。

参数：Format 参数常用的值如表 8-1。这是 Windows 和其他应用程序常常提供的格式。另外，该函数也支持定制的格式，只不过自行定义的格式需要进行登记注册后才能使用，这将在后文中进行阐述。



跟我来

1. 文本

在 Delphi 中，对于文本类的元件本身都已经封装了对剪贴板的操作。例如 Tmemo 元件本身含有与剪贴板对象进行交互的方法，它们是 CutToClipboard（）、CopyToClipboard（）和 PasteFromClipboard（），分别用来进行剪切、拷贝和粘贴。为了能够对所选择的文本进行操作，除了用手工选择文本之外，还可以使用 Tmemo 元件的 SelectAll 和 ClearSelection 等方法进行选择。

如图 8-1 所示，程序中加入了三个按钮 ButtonCut、ButtonCopy 和 ButtonPaste，其单击事件分别用于剪切、拷贝和粘贴数据。元件 Memo 用于显示操作的数据。清单 8-1 中列出了剪切、拷贝和粘贴的处理过程。

清单 8-1　文本剪切、拷贝和粘贴

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls,clipbrd; //注意在该语句中包含 clipbrd 单元

type
  TFrmMain = class（TForm）
    Memo: TMemo;
    ButtonCut: TButton;
    ButtonCopy: TButton;
    ButtonPaste: TButton;
    procedure ButtonCutClick（Sender: TObject）;
    procedure ButtonCopyClick（Sender: TObject）;
    procedure ButtonPasteClick（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.ButtonCutClick（Sender: TObject）;
begin
    Memo.SelectAll;      //选择 Memo 中的所有数据
    Memo.CuttoClipboard;//将所选数据剪切到剪贴板中
end;

procedure TFrmMain.ButtonCopyClick（Sender: TObject）;
begin
```

```
    Memo.SelectAll;
    Memo.CopytoClipboard;  //将所选数据拷贝到剪贴板中
end;

procedure TFrmMain.ButtonPasteClick（Sender: TObject）;
begin
      Memo.PasteFromClipboard; //将剪贴板中数据粘贴至 Memo 中文本的当前位置处
end;

end.
```

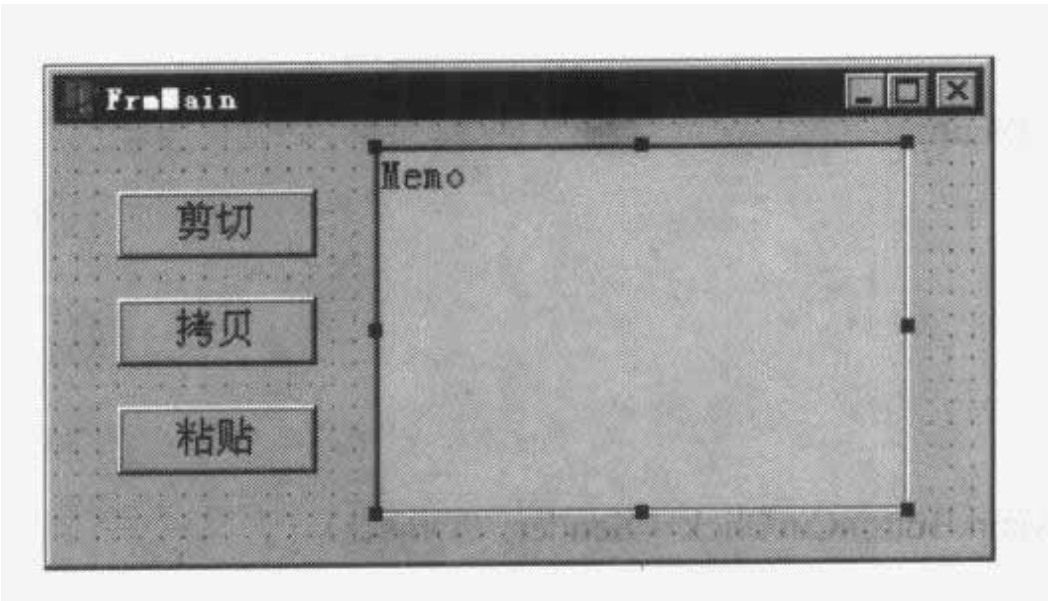


图 8-1　文本剪切、拷贝和粘贴

2. 图像

前面已经说过，剪贴板中也可以存放图形数据。现在对上述程序进行一下改造，使它除了能够操作文本数据外，还可以操作图形数据。具体操作哪种格式，将取决于剪贴板格式。在原有 Form 中加入 Timage 控件，名称为 Image。具体代码见清单 8-2。

清单 8-2　文本、图像数据的拷贝和粘贴

```
unit main;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
   ExtCtrls, StdCtrls,clipbrd; //注意在该语句中包含 clipbrd 单元

type
   TFrmMain = class（TForm）
      Memo: TMemo;
      ButtonCut: TButton;
      ButtonCopy: TButton;
      ButtonPaste: TButton;
```

```
    Image: TImage;
    procedure ButtonCutClick（Sender: TObject）;
    procedure ButtonCopyClick（Sender: TObject）;
    procedure ButtonPasteClick（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.ButtonCutClick（Sender: TObject）;
begin
      Clipboard.Assign（image.picture.bitmap）; //将图形数据复制至剪贴板中
      Image.Picture.Assign（Nil）; //清除当前数据
end;

procedure TFrmMain.ButtonCopyClick（Sender: TObject）;
begin
      Clipboard.Assign（Image.picture.bitmap）; //将图形数据复制至剪贴板中
end;

    procedure TFrmMain.ButtonPasteClick（Sender: TObject）;
    begin
        if Clipboard.HasFormat（CF_BITMAP） then  //如果剪贴板格式为位图格式则放
入 Image 中
            Image.Picture.Bitmap.Assign（Clipboard）
        else
        if Clipboard.HasFormat（CF_TEXT） then //如果剪贴板格式为文本格式则放入
Memo 中
            Memo.PasteFromClipboard;
    end;


    end.
```

在清单 8-2 中，ButtonCopy.OnClick 事件中调用了 Clipboard.Assign （）方法来把图像复制到剪贴板中。这样，其他应用程序就可以从中粘贴图像，只要它们支持 CF_BITMAP 格式就行。

另外，在 ButtonPaste.OnClick 事件中还调用了 Clipboard.HasFormat（）方法来判断剪贴板中的格式是否为指定的剪贴板格式，如果为图像格式则调用 Image.Picture.Bitmap.Assign （）方法将剪贴板中数据复制到 Image 元件的 Picture 中。如在粘贴之前，已经分别拷贝了文本和图像数据，那么在按“粘贴”按钮后，将出现如图 8-2 的调用界面。



图 8-2　文本、图像数据的拷贝和粘贴



再送礼包

正如清单 8-2 所示，只需要调用 Tmemo 的方法就能操纵剪贴板。当然，也可以通过剪贴板的 Clipboard.Astext 属性来手工地把文本放到剪贴板中。在 16 位 Windows 环境下，Astext 也许最多只能达到 255 个字符，因此更长的字符串只能调用 SetTextBuf（）和 GetTextBuf（）来实现。而在 32 位 Windows 环境下，这个问题就不存在了，因为在该环境中 AsText 是长字符串类型。不过，SetTextBuf（）和 GetTextBuf（）仍然是可用的。



自动感知剪贴板操作



遇到难题

现在我们对剪贴板已经有了一定的了解，但我们在多个应用程序间使用剪贴板时会发现，当在其中一个程序中使用了拷贝功能，则在另一个应用程序中的编辑菜单中，剪切和拷贝菜单项就会变成灰色（不可选），而粘贴菜单项则自动变成有效状态。那么，程序中是怎么自动感知剪贴板的剪切和拷贝操作的呢？



钥匙在此

为了使应用程序能自动感知剪贴板的更改，首先应当将该应用程序的窗体注册在剪贴板观测链（the clipboard viewer chain）中，然后使应用程序能响应剪贴板的更改消息，并对这些消息进行处理，最终还能将该窗体从链中移去。

1. 在剪贴板观测链中注册窗体

为了在剪贴板的观测链中注册应用程序的窗体，需要调用 Win32 API 函数 SetClipboardViewer（），该函数定义如下：

格式：function SetClipboardViewer（hWndNewViewer: HWND）: HWND;

功能描述：该函数将指定的窗体 hWndNewViewer 加入到剪贴板观测链中。

参数：hWndNewViewer 为要注册的窗体句柄。

返回值：如果注册成功，则返回值表示剪贴板中的下一个窗体的句柄。如果发生错误，或者在剪贴板观测链中已无其他窗体，则函数返回值为 NULL.。

提示：为了获得错误信息可以调用 GetLastError 系统函数。

2. 定义剪贴板更改消息的处理函数

如果剪贴板发生变化，则 Windows 系统将向剪贴板观测链中的窗体发送 WM_CHANGECBCHAIN 或者 WM_DRAWCLIPBOARD 消息，链中的每个窗体都会调用 SendMessage 函数将该消息传送给下一个窗体。为了能使程序自动感知剪贴板的更改消息，应该定义相应的消息处理函数来实现。

3. 将窗体从链中移去

链中的窗体最终会从剪贴板观测链中移去的，移去剪贴板观测链中的窗体时应调用 ChangeClipboardChain（）函数。其定义如下：

格式：function ChangeClipboardChain（hWndRemove, hWndNewNext: HWND）: BOOL;

功能描述：将窗体 hWndRemove 从剪贴板的观测链中移去。

参数：hWndRemove 为要移去的窗体句柄；hWndNewNext 标识了链中跟随于 hWndRemove 之后的窗体句柄（该值为 SetClipboardViewer 函数返回的窗体句柄）。

返回值：如果移去成功，则返回为 True。



跟我来

通过对清单 8-2 进行改进，使之能自动感知拷贝或剪切操作。改进的代码见清单 8-3。

清单 8-3　自动感知剪贴板操作

```
unit main;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls,clipbrd; //注意在该语句中包含 clipbrd 单元

type
  TFrmMain = class（TForm）
    Memo: TMemo;
    ButtonCut: TButton;
    ButtonCopy: TButton;
    ButtonPaste: TButton;
    Image: TImage;
    procedure ButtonCutClick（Sender: TObject）;
    procedure ButtonCopyClick（Sender: TObject）;
    procedure ButtonPasteClick（Sender: TObject）;
    procedure FormShow（Sender: TObject）;
    procedure FormClose（Sender: TObject; var Action: TCloseAction）;
  private
    { Private declarations }
  public
    //剪贴板观测链中的下一个窗体句柄
    NextClipViewHdle:HWND;
    //WM_DRAWCLIPBOARD 消息处理函数
    procedure WMDrawClipboard ( var MSG:TWMDrawClipboard ) ;message
WM_DRAWCLIPBOARD;
  end;

var
  FrmMain: TFrmMain;

implementation

{$R *.DFM}

procedure TFrmMain.ButtonCutClick（Sender: TObject）;
begin
    Clipboard.Assign（image.picture.bitmap）; //将图形数据复制到剪贴板中
    Image.Picture.Assign（Nil）; //清除当前数据
end;

procedure TFrmMain.ButtonCopyClick（Sender: TObject）;
```

```
begin
    Clipboard.Assign（Image.picture.bitmap）; //将图形数据复制到剪贴板中
end;

procedure TFrmMain.ButtonPasteClick（Sender: TObject）;
begin
    if Clipboard.HasFormat（CF_BITMAP） then begin //如果剪贴板格式为位图格式
则放入 Image 中
        Image.Picture.Bitmap.Assign（Clipboard）;
        ButtonPaste.Enabled:=false;
    end
    else
    if Clipboard.HasFormat（CF_TEXT） then //如果剪贴板格式为文本格式则放入
Memo 中
    begin
        Memo.PasteFromClipboard;
        ButtonPaste.Enabled:=false;
    end;
end;

procedure TFrmMain.WMDrawClipboard（var MSG:TWMDrawClipboard）;
begin
    if Clipboard.HasFormat（CF_TEXT） or Clipboard.HasFormat（CF_BITMAP ）
then
        ButtonPaste.Enabled:=true
    else
        ButtonPaste.Enabled:=false;
end;

procedure TFrmMain.FormShow（Sender: TObject）;
begin
    if Clipboard.HasFormat（CF_TEXT） or Clipboard.HasFormat（CF_BITMAP ）
then
        ButtonPaste.Enabled:=true
    else
        ButtonPaste.Enabled:=false;
    //将本窗体注册到剪贴板观测链中
    NextClipViewHdle:=SetClipboardViewer（handle）;
end;
```

```
procedure TFrmMain.FormClose（Sender: TObject; var Action: TCloseAction）;
begin
    //将本窗体从剪贴板观测链中移去
    ChangeClipboardChain（Handle,NextClipViewHdle）;
end;

end.
```

运行上述代码后，如果剪贴板中无文本或位图信息时，则“粘贴”按钮呈灰色。如果剪贴板中有文本或位图信息，则“粘贴”按钮将变成有效状态。例如，假设剪贴板中无任何信息，此时若使用画笔进行图像信息的复制，则此程序界面中的“粘贴”按钮会自动变成有效状态，如果此时单击“粘贴”按钮，该按钮又会变成灰色，如图 8-3 所示。

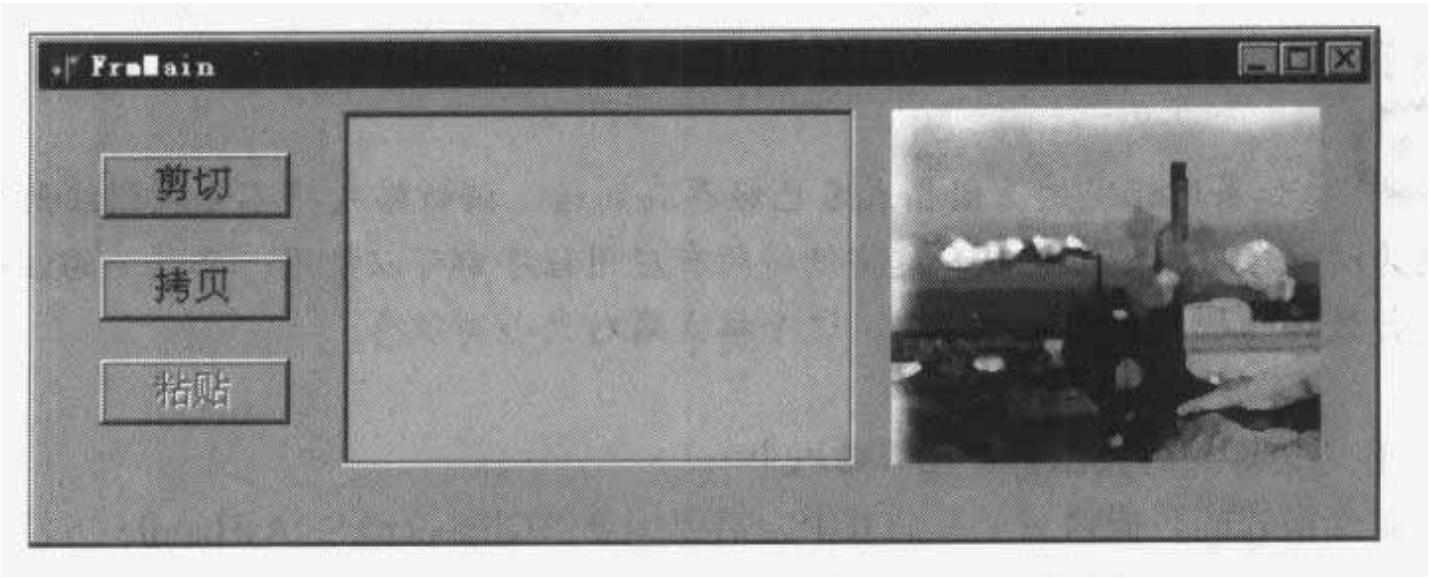


图 8-3　粘贴图像信息



创建自定义剪贴板格式



遇到难题

假如我们正在使用一个通信录程序，如果现在要输入的记录与前一条记录内容相差无几，此时你最好借助于剪贴板来把前一条记录的内容复制给当前的记录，而不是一个一个字段的录入。对于其他类型的应用程序，道理也是一样的。读者也许会问，该记录与前面所述的文本或者图像格式都是不一样的，而且记录中各字段的类型还不一样，要通过剪贴板来实现这些特殊格式的拷贝和粘贴，该如何来自定义剪贴板呢？

下面将创建一个对象，通过它来把特殊格式的数据保存到剪贴板中。



钥匙在此

拷贝和粘贴自定义的剪贴板格式数据包含了三个过程，即：如何注册自定义剪贴板格式；如何将自定义格式的数据拷贝至剪贴板中；又如何将剪贴板中的数据拷贝到自定义的数据结构中。

1. 登记注册自定义的剪贴板格式

在使用自定义的剪贴板之前，应先注册该剪贴板格式。在 Delphi 中，需要使用 Windows API 函数来实现这一目标。该函数为 RegisterClipboardFormat。

函数格式：function RegisterClipboardFormatA（lpszFormat: PAnsiChar）: UINT;

参数： lpszFormat 为新格式的字符串的指针。

返回值：如果函数执行成功，则将返回该格式的标识符；如果执行失败，则返回值为 0。可以调用系统函数 GetLastError 获取错误信息。



如果所指定的注册格式名已经存在的话，该新格式将不会得到注册登记，而返回值则为所存在格式的标识符。这将使得所有应用程序都可以使用一个统一的注册的剪贴板格式进行数据的拷贝和粘贴。注意，这个格式名对大小写敏感。

2. 将自定义格式的数据拷贝至剪贴板中

将自定义格式的数据拷贝至剪贴板中，需要用到 TClipboard.SetAsHandle 方法，该方法用于将句柄参数所指定的数据拷贝至剪贴板中。具体说明如下：

过程格式：procedure Tclipboard. SetAsHandle （Format: Word; Value: THandle）;

功能描述：SetAsHandle 将句柄 Value 所指定的数据写入格式为 Format 的剪贴板中。

参数：Format 参数可以指定为任一标准格式，也可以指定为自定义的格式。



在使用 SetAsHandle 传递句柄之后，请不要删除该句柄，因为该句柄将属于剪贴板，剪贴板将会删除它。

3. 从剪贴板中返回句柄给数据

该过程正好与 2 中的过程相反，需要调用的函数为 TClipboard.GetAsHandle。该函数说明如下：

函数格式：function TClipboard.GetAsHandle （Format: Word）: THandle;

功能描述：从指定格式的剪贴板中返回句柄给数据。用该方法可以重新从剪贴板中获得数据。

参数：Format 为要从中取数据的剪贴板格式的名称。

返回值：如果函数执行成功，则返回该格式的句柄。



GetAsHandle 返回的句柄，不仅仅被该应用程序所拥有，而且剪贴板一旦被打开就有效。如果需要长时间拥有它的话，可以将数据拷贝给另一个句柄。



跟我来

1. 创建一个剪贴板感知的对象

你可能会想到，自定义一种剪贴板格式的方法就是创建一个 Tclipboard 的派生类。这个特殊的类应当包含一些特殊的方法来处理特殊的格式。虽然这些格式可以解决一些特殊的问题，但是，一旦碰到其他数据格式时又得定义一种新的剪贴板格式。这无疑太麻烦了，而且也不是一个好的编程风格。

一个较好的方法是定义一个对象将数据封装起来，并使它对数据进行感知，即封装数据的拷贝和粘贴功能。例如，Delphi 的很多控件就封装了这些功能。Tmemo 元件知道如何将数据放到剪贴板中，Timage 元件也有方法将图形信息放到剪贴板中。实际上，这些元件最终都使用了 Tclipboard 对象，而且这样也不会引起冲突。

下面将采用这种“感知”的方法定义一种剪贴板格式。它是一个记录，其中包含了姓名、年龄和出生日期等字段。清单 8-4 列出了相关的代码。

清单 8-4　自定义一种剪贴板格式

```
unit cbdata;
interface
uses
  SysUtils, Windows, clipbrd;

const

  DDGData = 'CF_DDG'; //   用于注册剪贴板格式的常量
type

  // 存放在剪贴板中的数据格式
TDataRec = packed record
  Name: string[8];
  Age: Integer;
  BirthDate: TDateTime;
end;

{ 借助 TDataRec 定义对象 TData，该对象包含的两个方法分别用于对剪贴板进行
 拷贝和粘贴。 }
TData = class
public
  Rec: TDataRec;
```

```
    procedure CopyToClipBoard;
    procedure GetFromClipBoard;
  end;

var
  CF_DDGDATA: word; // 获得 RegisterClipboardFormat（）的返回值。

implementation

procedure TData.CopyToClipBoard;
{该函数用于拷贝属性 Rec 的值至剪贴板中，该属性中包含了二进制、文本数据格式。
}
const
  CRLF = #13#10;
var
  Data: THandle;
  DataPtr: Pointer;
  TempStr: String[50];
begin
  // 从数据堆中分配 SizeOf（TDataRec）字节的空间
  Data := GlobalAlloc（GMEM_MOVEABLE, SizeOf（TDataRec））;
  try
    // 获得所分配空间的第一个字节的地址指针
    DataPtr := GlobalLock（Data）;
    try
      // 将在 Rec 中的数据移入内存块中
      Move（Rec, DataPtr^, SizeOf（TDataRec））;
      { 如果多个剪贴板格式同时被拷贝入剪贴板中，则必须调用 Clipboard.Open，否
则只有一种格式被拷贝到剪贴板中。}
      ClipBoard.Open;
      // 将 Data 所代表的数据以 CF_DDGDATA 格式写入剪贴板中
      ClipBoard.SetAsHandle（CF_DDGDATA, Data）;
      // 以文本格式将数据拷贝至剪贴板中
      with Rec do
        TempStr := Name+CRLF+IntToStr（Age）+CRLF+
                   DateTimeToStr（BirthDate）;
      ClipBoard.AsText := TempStr;
      { 如果调用了 Clipboard.Open，为了与前面匹配，也应调用 Clipboard.Close }
      Clipboard.Close
    finally
```

```
        // 释放所分配的内存
        GlobalUnlock（Data）;
      end;
    except
      GlobalFree（Data）;
      raise;
    end;
  end;

  procedure TData.GetFromClipBoard;
  { 这个方法用于粘贴存放在剪贴板中的格式为 CF_DDGDATA 的数据。这个数据将存放在这个对象的 TDataRec 域。}
  var
    Data: THandle;
    DataPtr: Pointer;
    Size: Integer;
  begin
    // 获得剪贴板的句柄
    Data := ClipBoard.GetAsHandle（CF_DDGDATA）;
    if Data = 0 then Exit;
    // 获得指向句柄为 Data 的内存块的指针
    DataPtr := GlobalLock（Data）;
    try
      // 重新获得该数据的大小
      if SizeOf（TDataRec） > GlobalSize（Data） then
        Size := GlobalSize（Data）
      else
        Size := SizeOf（TDataRec）;
      // 将数据拷贝到 TDataRec 域中
      Move（DataPtr^, Rec, Size）
    finally
      // 释放内存
      GlobalUnlock（Data）;
    end;
  end;

  initialization
    // 登记注册自定义的剪贴板格式
    CF_DDGDATA := RegisterClipBoardFormat（DDGData）;
  end.
```

该段程序主要做了以下几件事：

首先，它调用 RegisterClipBoardFormat（）注册了一个新的剪贴板格式，并返回这个格式的标识符。如果其他应用程序已经注册了同样的格式，则得到的返回值是相同的。新注册的格式将出现在 Clipboard.Formats 数组中。

其次，定义了一个对自定义格式感知的对象。在该对象中，包含了类型为 TdataRec 的域 Rec，并封装了 CopytoClipboard（）和 GetFromClipboard（）两个方法。

其中，TData. CopytoClipboard（）支持以 CF_DDGDATA 和 CF_TEXT 两种格式将域 Rec 的内容放到剪贴板中。不过，在调用 Clipboard.SetAsHandle（）之前，必须先准备好一个合法的 Thandle，以传给 SetAshandle（），这个句柄代表被放到剪贴板中的数据。TData.GetFromClipboard（）的作用正好相反，它以 CF_DDGDATA 格式从剪贴板中获取数据，并将其放到 Tdata.Rec 中。

下面要介绍的程序将显示如何调用这个方法。注意，只要做适当的修改，剪贴板对象就能支持其他自定义的记录类型。对句柄以及对数据堆的操作参见本节的“再送礼包”。

2. 使用自定义的剪贴板格式

下面介绍的程序将显示如何使用上述自定义的剪贴板格式。这个程序的主 Form 如图 8-4 所示。

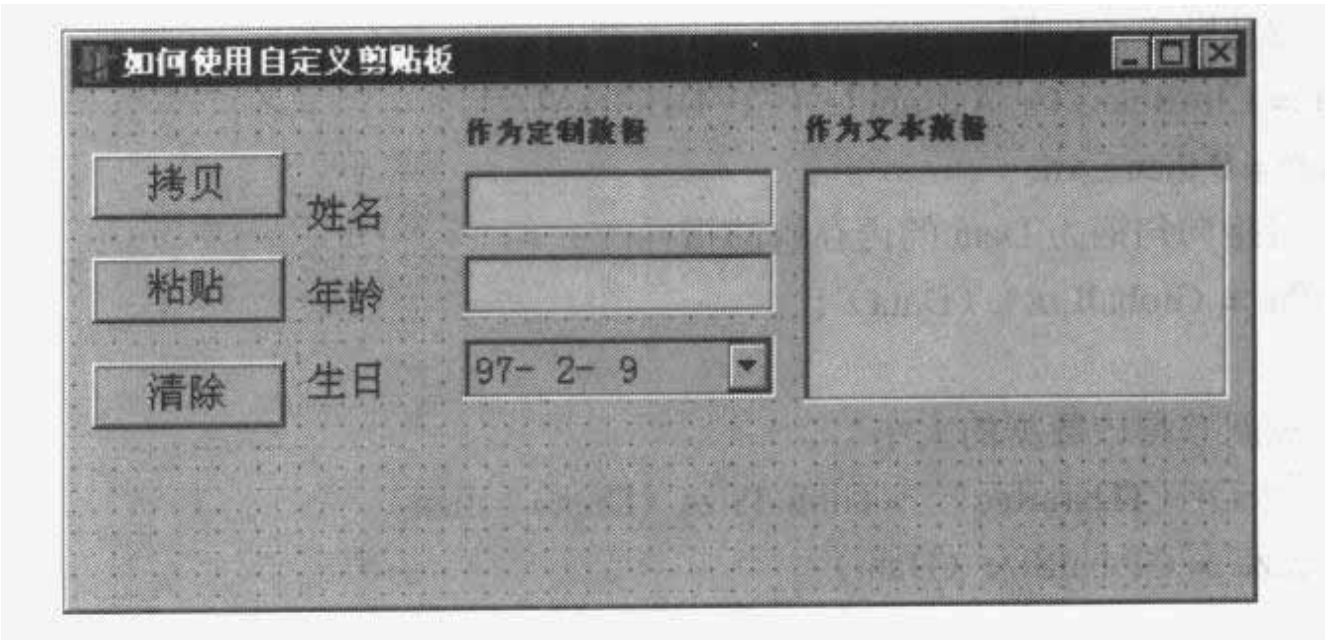


图 8-4　程序主界面

可以看出，这个窗体上的一些控件是用来填充 Tdata 对象的数据的。清单 8-5 列出了主窗体的源代码。

清单 8-5　使用自定义剪贴板格式的主窗体代码

```
unit MainFrm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, clipbrd, Mask, ComCtrls;
type
```

```
TMainForm = class（TForm）
    edtFirstName: TEdit;
    btnCopy: TButton;
    btnPaste: TButton;
    meAge: TMaskEdit;
    btnClear: TButton;
    lblFirstName: TLabel;
    lblAge: TLabel;
    lblBirthDate: TLabel;
    memAsText: TMemo;
    lblCustom: TLabel;
    lblText: TLabel;
    dtpBirthDate: TDateTimePicker;
    procedure btnCopyClick（Sender: TObject）;
    procedure btnPasteClick（Sender: TObject）;
    procedure btnClearClick（Sender: TObject）;
  end;

var
  MainForm: TMainForm;

implementation
uses cbdata;

{$R *.DFM}

procedure TMainForm.btnCopyClick（Sender: TObject）;
// 该方法将窗体中各控件的数据拷贝至剪贴板中
var
  DataObj: TData;
begin
  DataObj := TData.Create;
  try
    with DataObj.Rec do
    begin
      Name      := edtFirstName.Text;
      Age       := StrToInt（meAge.Text）;
      BirthDate := dtpBirthDate.Date;
      DataObj.CopyToClipBoard;
```

```
    end;
  finally
    DataObj.Free;
  end;
end;

procedure TMainForm.btnPasteClick（Sender: TObject）;
{ 该方法将 CF_DDGDATA 格式的数据从剪贴板中粘贴至窗体的控件中，
  将文本格式的剪贴板格式拷贝至 TMemo 组件中。
  }
var
  DataObj: TData;
begin
  btnClearClick（Nil）;
  DataObj := TData.Create;
  try
    // 检查 CF_DDGDATA 格式是否可用
    if ClipBoard.HasFormat（CF_DDGDATA） then
      // 将 CF_DDGDATA 格式的数据拷贝至窗体的各控件中
      with DataObj.Rec do
      begin
        DataObj.GetFromClipBoard;
        edtFirstName.Text := Name;
        meAge.Text        := IntToStr（Age）;
        dtpBirthDate.Date := BirthDate;
      end;
  finally
    DataObj.Free;
  end;
  // 将文本格式的数据粘贴至 TMemo 组件中
  if ClipBoard.HasFormat（CF_TEXT） then
    memAsText.PasteFromClipBoard;
end;

procedure TMainForm.btnClearClick（Sender: TObject）;
var
  i: integer;
begin
  // 清除所有控件中的数据内容
  for i := 0 to ComponentCount - 1 do
```

```
        if Components[i] is TCustomEdit then
          TCustomEdit（Components[i]）.Text := '';
    end;


    end.
```

运行程序，如果用户单击“拷贝”按钮就将把 Tedit 和 TdatatimePicker 控件中的数据复制给 Tdata 对象的 TdataRec 域，并调用 Tdata.CopyToClipboard（）将数据放到剪贴板中。

如果用户单击“粘贴”按钮，则情况相反。首先，如果剪贴板中的数据为 CF_DDGDATA 格式，则从剪贴板中粘贴数据，并把它放到 Form 的编辑框中，同时，字符串形式的数据被放到 Form 的 Tmemo 元件中。粘贴操作的结果如图 8-5 所示。你也可以将文本粘贴到其他 Windows 程序中，譬如记事本 Notepad 中。

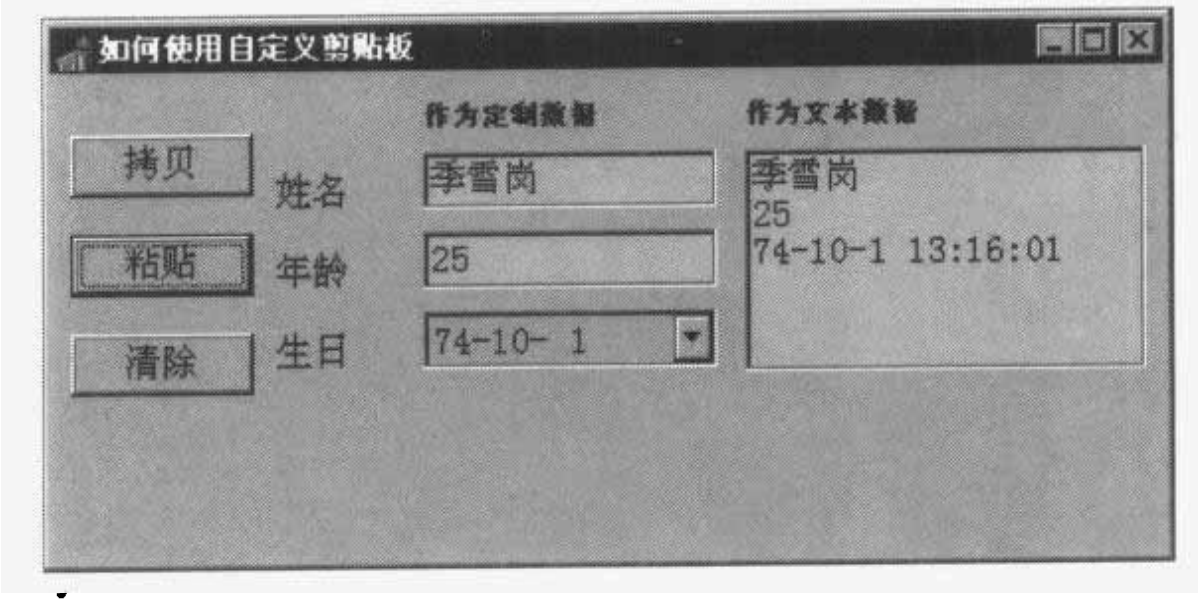


图 8-5　粘贴的数据

单击“清除”按钮，可以清空主 Form 上所有控件的内容。



再送礼包

1. 句柄

一个句柄实际上就是一个 32 位的变量，不过它的值是由操作系统进行统一调配的，它代表着 Win32 管理的一个内存块。句柄的类型有许多种，如 Delphi 的 TForm，Tiamge，Tbitmap，TCanvas 中都封装了句柄。

2. 有关数据堆的操作

有些 Win32 函数，如剪贴板函数，使用堆来操作数据。为了访问堆，需要调用下列函数：

GlobalAlloc（）	在堆中分配若干个字节并返回相应的句柄；
GlobalFree（）	释放由 GlobalAlloc（）分配的堆；
GlobalLock（）	返回由 GlobalAlloc（）分配的堆的指针；
GlobalUnlock（）	解除 Globalock（）对堆的锁定。



总结

为了共享自己独有的信息，自定义剪贴板格式可以为应用程序发挥很大的作用，同时可以为程序的使用人员带来很大的方便。

第九章

串口通信

热点透视

随着现代信息技术的发展以及计算机网络的广泛使用，计算机通信技术已经日臻成熟，但串行通信作为一种灵活方便可靠的通信方式，仍不失为有效的通信手段，因此它被广泛应用于工业控制中。Delphi 中并未提供现成的串口控件供程序员使用，这是因为 WinAPI 已经提供了一个比较容易使用的编程接口。本章将讲述：

- 如何通过串口配置对话框更改串口配置
- 如何实现串口通信



通过串口配置对话框更改串口配置



遇到难题

如果应用程序支持串口通信的话，肯定需要对串口的硬件参数进行合理的配置，例如，波特率、数据位、流量控制等，这种配置工作往往是留给用户进行的。我们知道，在控制面板的系统设备中可以配置这一端口，但有时需要将这一配置功能加入应用程序中。您也许会认为在自己的应用程序中又得重新开发一个串口配置对话框了，其实根本不用重新设计这个对话框，而只需在应用程序中进行调用即可。那么如何实现这一功能呢？



钥匙在此

如果想通过串口配置对话框来更改串口配置的话，则需要调用 Windows API 函数来实现。首先要调用 CreateFile（）函数来打开串口，然后调用 GetCommConfig（）函数来获取当前串口配置的结构信息，接着调用 CommConfigDialog（）弹出配置对话框由用户进行串口参数的配置，如果用户按了该对话框的“确定”按钮的话，则由 SetCommConfig（）函数根据用户所输入的值来设置该串口，最后不要忘了关闭串口的句柄。下面，分别讨论这几个 API 函数。

1. 打开串口

可以通过调用 CreateFile（）函数来创建和打开串口，并且返回串口的句柄。当然，该函数还用于创建和打开其他对象，如文件、管道、磁盘设备等等，本文要阐述的则是串口对象。该函数在 Windows.pas 单元中是如下声明的：

```
function CreateFile（lpFileName: PChar; dwDesiredAccess, dwShareMode: DWORD;
   lpSecurityAttributes: PSecurityAttributes; dwCreationDisposition,
dwFlagsAndAttributes: DWORD;   hTemplateFile: THandle）: THandle; stdcall;
```

其中，参数 lpFileName 指定要操作的对象名称，对于串口而言，则是串口的名称，如“COM1”表示串口 1。同样，如果还有其他串口则应指定相应的名称。

参数 dwDesiredAccess 表示要访问该对象的类型。一个应用程序可以对串口进行读操作、写操作、读写操作以及进行设备查询操作等。该参数可以取以下值，或者是这些值的组合，它的取值见表 9-1。

表 9-1　　dwDesiredAccess 取值

值	含 义
0	表示对设备进行查询访问。应用程序可以只查询设备属性而不进行设备访问
GENERIC_READ	表示对操作对象进行读操作。数据可以从文件以及可以被移动的文件指针中读取。可以组合 GENERIC_WRITE 进行读写操作
GENERIC_WRITE	表示对操作对象进行写操作。数据可以向文件以及可以被移动的文件指针中写入。可以组合 GENERIC_READ 进行读写操作

参数 dwShareMode 是指定对象如何被共享的位标志组。如果 dwShareMode 为 0，则对象不能被共享，于是后来的打开该对象的操作将失败，直到该对象句柄被关闭为止。为了共享该对象，可以使用一个或多个下列值，具体值见表 9-2。

表 9-2　　dwShareMode 取值

值	含 义
FILE_SHARE_DELETE	只用于 Windows NT，如果请求了删除访问，则后续的打开操作将成功
FILE_SHARE_READ	如果请求了读取访问，则后续的打开操作将成功
FILE_SHARE_WRITE	如果请求了写访问，则后续的打开操作将成功

参数 lpSecurityAttributes 表示指向 SECURITY_ATTRIBUTES 结构变量的指针。该结构确定返回的对象句柄是否可以被其子线程所继承使用。如果该参数被设为空，则表明句柄不能被继承。

参数 dwCreationDistribution 表示对已经存在的文件怎样进行处理，以及当文件不存在的时候又进行什么样的处理。该参数必须为下列值之一，见表 9-3。

表 9-3　　dwCreationDistribution 取值

值	含 义
CREATE_NEW	创建一个新文件。如果指定的文件已经存在则该函数失败
CREATE_ALWAYS	创建一个新文件。如果文件存在，则进行覆盖
OPEN_EXISTING	打开一个文件。如果文件不存在，则函数失败
OPEN_ALWAYS	如果文件存在，则打开文件。如果文件不存在，则会新建一个文件
TRUNCATE_EXISTING	打开一个文件。一旦文件被打开，文件被缩短以使它的大小为 0

参数 dwFlagsAndAttributes 指定文件的属性以及文件的标志。任何一个下列属性的组合都是合理的。

表 9-4　　dwFlagsAndAttributes 取值

属 性	含 义
FILE_ATTRIBUTE_ARCHIVE	该文件为文档文件
FILE_ATTRIBUTE_COMPRESSED	目录或文件是压缩的
FILE_ATTRIBUTE_HIDDEN	文件是隐藏的
FILE_ATTRIBUTE_NORMAL	正常文件

续表

属 性	含 义
FILE_ATTRIBUTE_OFFLINE	文件的数据不能立即使用。意味着文件数据可能被物理地移走了
FILE_ATTRIBUTE_READONLY	只读文件
FILE_ATTRIBUTE_SYSTEM	系统文件
FILE_ATTRIBUTE_TEMPORARY	临时文件

参数 hTemplateFile 在 Windows 95 中应为空。

下例用于打开串口 COM1，显示了该函数的使用方法。

```
var
  CommPort : string;
  hCommFile : THandle;
begin
  CommPort := 'COM1';
 {打开指定串口}
  hCommFile := CreateFile（PChar（CommPort），
                          GENERIC_WRITE,
                          0,
                          Nil,
                          OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL,
                          0）；
end;
```

2. 获取串口配置信息

为了获取串口的配置信息，需要调用 GetCommConfig（），该函数在 Windows.pas 单元文件中声明如下：

function GetCommConfig（hCommDev: THandle; var lpCC: TCommConfig; var lpdwSize: DWORD）: BOOL; stdcall;

该函数用于得到当前通信设备的配置信息。其中参数 hCommDev 表示所打开的通信设备句柄。LpCC 表示存放配置信息的缓冲区，其数据结构为 TcommConfig。LpdwSize 为缓冲区的大小。其调用方法如下：

```
GetCommConfig（hCommFile, Buffer^, size）；
```

3. 显示串口配置对话框

为了在应用程序中弹出驱动程序所提供的串口配置对话框，需要调用 CommConfigDialog（）函数。该函数在 windows.pas 单元文件中的声明格式如下：

function CommConfigDialog（lpszName: PChar; hWnd: HWND; var lpCC: TCommConfig）: BOOL; stdcall;

其中，参数 lpszName 表示对话框配置的设备名称。HWnd 为拥有对话框的窗体句柄。参数 lpCC 表示指向配置参数结构 TcommConfig 的指针。如下例所示：

```
if CommConfigDialog（PChar（CommPort），
```

```
                    Form1.Handle,
                    Buffer^）  = true then begin
    {如果用户按“确定”按钮，则根据输入的值设置串口}
     SetCommConfig（hCommFile, Buffer^, size）;
  end;
```

4. 设置串口

为了根据所输入的值对串口参数进行设置，需要调用 SetCommConfig（）函数。

function SetCommConfig（hCommDev: THandle; const lpCC: TCommConfig; dwSize: DWORD）: BOOL; stdcall;

该函数用于设置通信设备的当前配置。其中，参数 hCommDev 为要打开的设备的句柄；参数 lpCC 为指向 TcommConfig 结构变量的指针；参数 dwSize 为该结构变量的大小。该函数的使用方法如下：

```
SetCommConfig（hCommFile, Buffer^, size）;
```



跟我来

如何对串口进行设置，可以参考下面这个完整的例子。新建应用程序，然后在窗体上放置一个按钮，完整代码参见清单 9-1。

清单 9-1　由配置对话框更改串口配置

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class（TForm）
    Button1: TButton;
    procedure Button1Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}

procedure TForm1.Button1Click（Sender: TObject）;
var
   CommPort : string;
   hCommFile : THandle;
   Buffer : PCommConfig;
   size : DWORD;
begin
 CommPort := 'COM1';
{打开指定串口}
 hCommFile := CreateFile（PChar（CommPort）,
                          GENERIC_WRITE,
                          0,
                          Nil,
                          OPEN_EXISTING,
                          FILE_ATTRIBUTE_NORMAL,
                          0）;
 if hCommFile=INVALID_HANDLE_VALUE then
 begin
   ShowMessage（'Unable to open '+ CommPort）;
   exit;
 end;
{获得一个临时缓冲区}
 GetMem（Buffer, sizeof（TCommConfig））;
{获取串口配置结构变量的大小}
 size := 0;
 GetCommConfig（hCommFile, Buffer^, size）;
{释放临时缓冲区}
 FreeMem（Buffer, sizeof（TCommConfig））;
{获取 CommConfig 结构}
 GetMem（Buffer, size）;
 GetCommConfig（hCommFile, Buffer^, size）;

{弹出串口配置对话框}
 if CommConfigDialog（PChar（CommPort）,
                      Form1.Handle,
```

```
                                    Buffer^）  = true then begin
      {如果用户按“确定”按钮，则根据输入的值设置串口}
       SetCommConfig（hCommFile, Buffer^, size）;
    end;

  {释放缓冲区}
   FreeMem（Buffer, size）;

  {关闭串口}
   CloseHandle（hCommFile）;
end;
end.
```

运行该应用程序，当点击按钮时，程序将会弹出串口的设置对话框供用户进行设置，对话框如图 9-1 所示。如果用户按确定按钮，程序将该设置值赋给相应的串口。

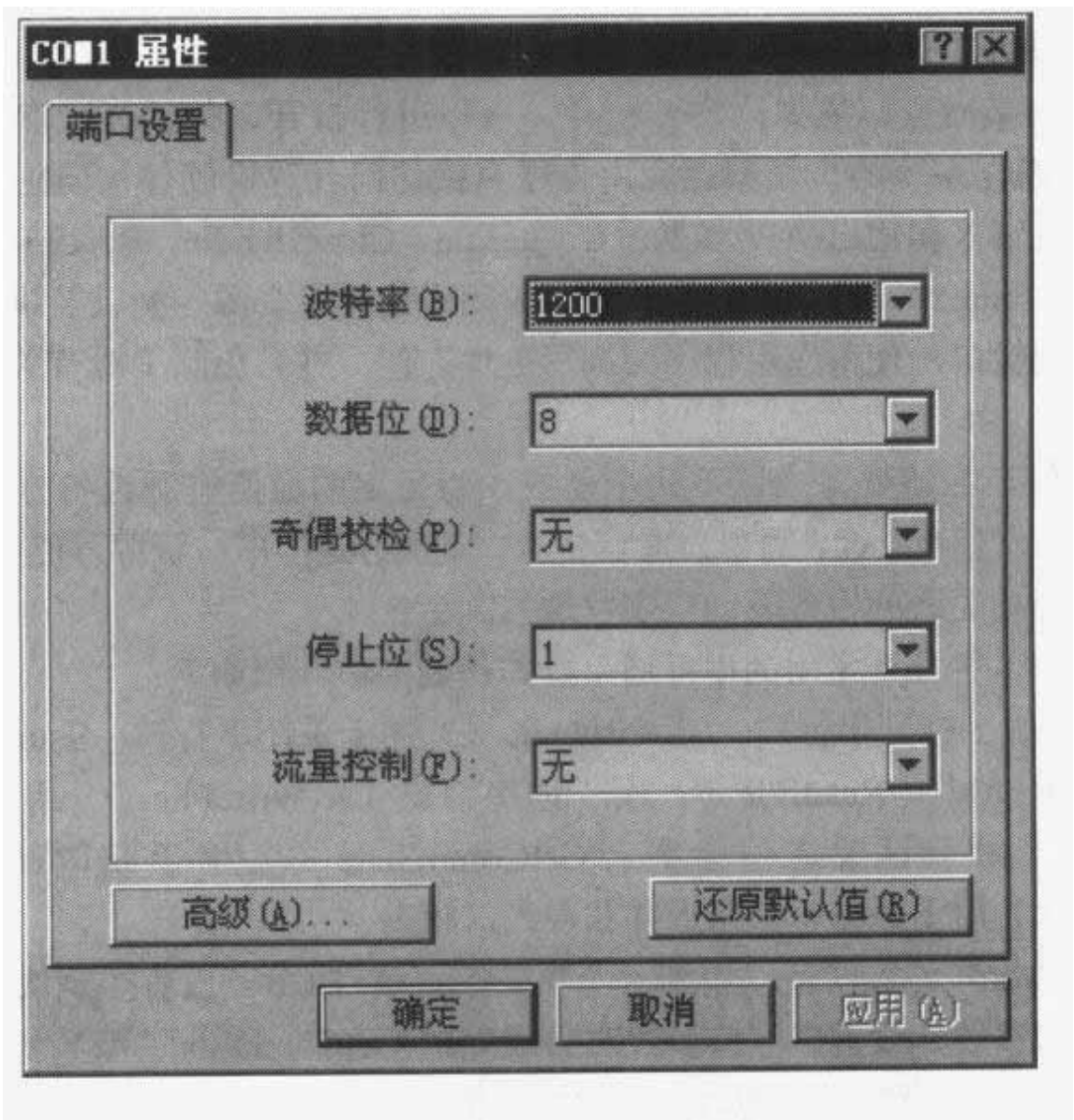


图 9-1　串口设置对话框



实现串口通信



遇到难题

串行通信作为一种灵活方便可靠的通信方式，被广泛应用于工业控制中。在工业生产实践中，如果用 PC 机对工程实行实时监控，通常要求 PC 机能在用户界面上具有数据采集、数据处理以及控制信号的产生与传输等功能。在这种特定的环境下，PC 机要与过程控制的实时信号相联系，于是要求能实现对 PC 机的串行端口进行直接操作。



钥匙在此

1. 基于 Win 95/NT 的串行通信机制

串口对于程序员而言，就如同文件一样，可以进行打开、关闭、读取数据、写数据等操作。Windows 操作系统的机制禁止应用程序直接访问计算机硬件，但它为程序员提供了一系列的标准文件输入和输出 API 函数（CreateFile、CloseHandle、ReadFile、ReadFileEx、WriteFile 和 WriteFileEx），这些函数为打开、关闭通信设备资源句柄以及执行读、写操作提供了基本和方便的接口，使得应用程序的编写更加方便，并且免除了对有关硬件进行调试的麻烦。

进行串口通信，应该要完成以下几个任务：为指定的通信资源设备打开句柄；设置和查询串口资源的配置信息；读、写串口通信资源；监视给定的串口资源可能发生的一组事件；向与指定设备资源相关联的设备驱动程序发送控制命令。

现将 Win 95/NT 中几个常用的串行通信操作函数简单介绍如下：

CreatFile（）用于打开串行口；CloseHandle（）用于关闭串行口；SetupComm（）用于设置通信缓冲区的大小；ReadFile（）用于读串口操作；WriteFile（）用于写串口操作；SetCommState（）用于设置通信参数；GetCommState（）用于获取缺省通信参数；ClearCommError（）用于清除串口错误并获取当前状态。

除上述几个函数外，还要经常用到一个重要的记录 DCB（设备控制块）。DCB 中记录有可定义的串行口参数，设置串行口参数时必须先用 GetCommState 函数将系统缺省值填入 DCB 控制块，然后才可设定用户想改变的自定义值。

在 Win 95/NT 中进行串行通信除了应该了解基本的通信操作函数外，还要掌握多线程编程技术。线程是进程内部执行的路径，是操作系统分配 CPU 时间的基本实体。每个进程都从单线程开始来完成应用程序的执行。串行通信需要利用多线程技术实现，其主要的处理逻辑可以表述如下：进程开始先由主线程做一些必要的初始化工作，然后主线程根据需要在适当时候建立通信监视线程监视通信口，当指定的串行口事件发生时，向主线程发送

WM_COMMNOTIFY 消息（由于 Win 95 取消了 WM_COMMNOTIFY 消息，因此必须自己创建），接着主线程对其进行处理。若不需要 WM_COMMNOTIFY 消息，则主线程终止通信监视线程。

当多线程同时执行的时候，将会引起对共享资源的冲突。为避免冲突，就要用同步多线程对共享资源进行访问。Win 95 提供了许多保持线程同步的方法。例如采用创建事件对象来保持线程同步时，先通过 CraeteEvent()创建事件对象，然后使用 SetEvent()或 PulseEvent（）函数将事件对象设置成信号同步，在应用程序中，利用 WaitSingleObject（）函数等待同步的触发，直到指定的事件被其他线程设置为有信号时，才继续向下执行程序。

2. Delphi 下的具体实现方法

Delphi 的强大功能和支持多线程的面向对象编程技术，使得实现串行通信非常简单方便。它通过调用外部的 API 函数来实现，主要步骤如下：首先，利用 CreateFile 函数打开串行口，以确定本应用程序对此串行口的占有权，并封锁其他应用程序对此串口的操作；其次，通过 GetCommState 函数填充设备控制块 DCB，再通过调用 SetCommState 函数配置串行口的波特率、数据位、校验位和停止位。然后，创建串行口监视线程监视串行口事件。在此基础上就可以在相应的串口上进行数据的传输；最后，用 CloseHandle 函数关闭串行口。



跟我来

清单 9-2 为串口通信的完整代码。

清单 9-2　串口通信

```
unit comdemou;

interface

uses
   Windows, Messages, SysUtils, Classes,Graphics, Controls, Forms, Dialogs,
   StdCtrls;

const
      Wm_commNotify=Wm_User+1;//自定义消息常量
type
   TForm1 = class（TForm）
      procedure FormCreate（Sender: TObject）;
   private
      Procedure comminitialize;
      Procedure MsgcommProcess（Var Message:Tmessage）; Message Wm_commnotify;
      { Private declarations }
   public
```

```
    { Public declarations }
  end;

  //线程声明
  TComm=Class（TThread）
  protected
     procedure Execute;override;
  end;

var
  Form1: TForm1;
  hcom,Post_Event:Thandle;
  lpol:Poverlapped;
implementation

{$R *.DFM}

Procedure TComm.Execute; //线程执行过程
var
   dwEvtMask:Dword;
   Wait:Boolean;
Begin
      fillchar（lpol,sizeof（toverlapped）,0）;
      While True do Begin
       dwEvtMask:=0;
       Wait:=WaitCommEvent（hcom,dwevtmask,lpol）;
       //等待串行口事件
       if Wait Then Begin
           waitforsingleobject（post_event,infinite）;
           //等待同步事件置位
           resetevent（post_event）;  //同步事件复位
           PostMessage（Form1.Handle,WM_COMMNOTIFY,0,0）;//发送消息
        end;
      end;
  end;

  procedure Tform1.comminitialize;
  //串行口初始化
  var
  lpdcb:Tdcb;
```

```
Begin
    //打开串口 2，用于读和写串口
    hcom:=createfile（'com2',
                     generic_read or generic_write,
                     0,
                     Nil,
                     open_existing,
                     file_attribute_normal or   file_flag_overlapped,
                     0）;//打开串行口
    if hcom=invalid_handle_value then begin
        ShowMessage（'Unable to open COM2'）;
        exit;
        end;
        setupcomm（hcom,4096,4096）;
        //设置输入和输出缓冲区皆为 4096 字节
        getcommstate（hcom,lpdcb）;
        //获取串行口当前缺省的设置
        lpdcb.baudrate:=2400;
        lpdcb.StopBits:=1;
        lpdcb.ByteSize:=8;
        lpdcb.Parity:=EvenParity;        //偶校验
        Setcommstate（hcom,lpdcb）;
        setcommMask（hcom,ev_rxchar）;
        //指定串行口事件为接收到字符
end;

Procedure TForm1.MsgcommProcess（Var Message:Tmessage）;
var
    Clear:Boolean;
    Coms:Tcomstat;
    cbNum,ReadNumber,lpErrors:DWord;
    Read_Buffer:array[1..100]of char;
Begin
    Clear:=Clearcommerror（hcom,lpErrors,@Coms）;
    if Clear Then Begin
        cbNum:=Coms.cbInQue;
        ReadFile（hCom,Read_Buffer, cbNum,ReadNumber,lpol）;
        //处理接收数据
        SetEvent（Post_Event）;
       //同步事件置位
```

```
    end;
end;

procedure TForm1.FormCreate（Sender: TObject）;
begin
    //串口初始化
    comminitialize;
    post_event:=CreateEvent（Nil,true,true,Nil）; //创建同步事件
    Tcomm.Create（False）;//创建串行口监视线程
end;

end.
```

该程序代码中，在窗体创建时先将串口初始化，打开串口并设置初始值（当然，如果不使用这些缺省值的话，可以使用前一节介绍的方法，弹出一个对话框让用户进行串口值的设置）。然后创建同步事件，接着再创建串口的监视线程。当系统串口接收到数据时将通知该应用程序，然后由 MsgcommProcess 消息函数负责对数据的接收和处理。

第十章

打印机与打印

热点透视

为了能较好的将自己的数据输出到打印机中，或者在应用程序中需要进行一些特殊的打印时，就应该对打印机的控制以及如何读取打印机的设置信息有一个较深的了解。Delphi 中使用 Tprinter 对象封装了打印接口，但要进行一些底层的控制或者需要读取一些驱动器中的信息时，VCL 也就无能为力了，这时需要调用 API 函数来实现。本章内容主要有：

- 指定当前要使用的打印机
- 获取和设置打印机基本信息
- 控制打印机设置
- 获取打印机详细信息



指定当前要使用的打印机



遇到难题

虽然可以通过打印管理器来指定缺省的打印机，但程序员可能想在程序运行时指定程序中要使用的打印机。例如，假如应用程序是用来绘图的，需要通过绘图仪来输出，而用户机器上装备了普通打印机和绘图仪，用户的其他文档处理是用的普通打印机，如果让用户为了打印而来回的进行修改缺省打印机的话势必给用户带来麻烦。因此，在应用程序中应该可以直接指定要使用的打印机。



钥匙在此

要指定某台打印机供应用程序使用需要完成两件事，首先应该获取所有已经安装的打印机的列表，然后将用户选择的打印机作为当前要使用的打印机。

为了获取所有已经安装的打印机的列表，可以直接访问 Tprinter.Printers 属性，该属性是一个字符串列表类型，即 Tstrings 类型，所以要获得可用的打印机列表的名称相当容易。

可以通过 Tprinter.PinterIndex 属性来确认打印机列表 Printers 属性中所选择的打印机。因此，可以通过该属性来访问当前所选择的打印机的名称，如语句

```
Tprinter.Printers[Tprinter.PrinterIndex]
```

返回的就是当前所选择的打印机的名称。

如果将 Tprinter.PrinterIndex 设为-1，则表示 printer 使用的是系统中缺省的打印机。



跟我来

新建一个应用程序，并在窗体上放置一个 Tcombobox 控件和一个 Tlabel 控件。清单 10-1 显示了如何指定缺省的打印机。

清单 10-1　指定缺省打印机

```
unit MainFrm;

interface
```

```
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TMainForm = class（TForm）
    cbPrinters: TComboBox;
    lblPrinter: TLabel;
    procedure FormCreate（Sender: TObject）;
    procedure cbPrintersChange（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation
uses IniFiles, Printers;

{$R *.DFM}

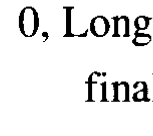
procedure TMainForm.FormCreate（Sender: TObject）;
begin
  { 将当前所有可用的打印机复制到组合框中
    并显示当前已经选中的缺省打印机}
  cbPrinters.Items.Assign（Printer.Printers）;
  cbPrinters.Text := Printer.Printers[Printer.PrinterIndex];
  // 将当前打印机名称显示到标签上
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

procedure TMainForm.cbPrintersChange（Sender: TObject）;
var
  IniFile: TIniFile;
  TempStr1, TempStr2: String;
  S: array[0..64] of char;
begin
```

```
with Printer do
begin
  // 将 ComboBox 中所选择的打印机作为缺省的打印机
  PrinterIndex := cbPrinters.ItemIndex;
  //将打印机的名字存放在临时字符串中
  TempStr1 := Printers[PrinterIndex];
  // 删除打印机名称中不必要的部分
  System.Delete（TempStr1, Pos（' on ', TempStr1）, Length（TempStr1））;
  // 创建 TIniFile 实例
  IniFile := TIniFile.Create（'WIN.INI'）;
  try
    // 获取所选择的打印机的设备名称
    TempStr2 := IniFile.ReadString（'Device', TempStr1, ''）;
      // 改变缺省打印机的名称
      IniFile.WriteString（'windows', 'device', TempStr1 + ',' + TempStr2）;
      // 向所有的窗体发送消息，说明设置已经发生了改变
      StrCopy（S, 'windows'）;
  SendMessage（HWND_BROADCAST, WM_SETTINGCHANGE ,
 0, LongInt（@S））;
    finally
      IniFile.Free;
    end;
  end;
  // 重新刷新标签显示
  lblPrinter.Caption := Printer.Printers[Printer.PrinterIndex];
end;

end.
```

在上述代码中，当窗体创建时，从 Printer.Printers 获取了所有的打印机的名称来填充到 Tcombobox 控件中，然后根据当前所选择的打印机来刷新 Tlabel 元件。事件句柄 cbPrintersChange()设置了缺省的打印机，这将使 windows 目录下的 win.ini 文件中的 window 字段中的 device 项发生变化。



再送礼包

另外，也可以使用 PrinterDialog 和 PrinterSetupDialog 控件来改变打印机和当前选中打印机的设置，如下所示：

```
PrinterDialog.execute;
```

或者

PrinterSetupDialog.Execute;

选择想要的一种方式。PrinterDialog 在打印一个页范围时更有用，而 PrinterSetupDialog 对改变纸盒等活动更为有用。两个打印机对话框都允许用户直接从对话框中选择打印机的属性。



获取和设置打印机基本信息



遇到难题

有时，在应用程序中只想简单的获取打印机的设备名称、驱动程序文件名以及打印机的安装端口名；或者，想将要打印的内容输出到文件中；有时，还要直接设置端口等。但是，如果要改变打印机的设置诸如纸盒来源、复制份数等，则本节内容就不适合了。



钥匙在此

要获取和设置打印机的一些简单信息，可以使用 Tprinter 的 GetPrinter（）和 SetPrinter（）两个方法。一般来说，这两个方法均不推荐使用且也没有很好的文档资料。但是如果只是简单地设置打印机或获得打印机的基本信息的话，还是比较适合使用的。

Printer.GetPrinter（）的声明格式如下：

GetPrinter （ADevice, ADriver, APort: PChar; var ADeviceMode: THandle）；

该方法的参数分别表示设备的名称、驱动文件名、端口号名以及指向打印机的 DevMode 结构的句柄变量。看下面这个例子是如何使用 GetPrinter（）方法的。

```
var
   pDevice : pChar;
   pDriver : pChar;
   pPort    : pChar;
   hDMode : THandle;
begin
      GetMem（pDevice, cchDeviceName）；
      GetMem（pDriver, MAX_PATH）；
      GetMem（pPort, MAX_PATH）；
      Printer.GetPrinter（pDevice, pDriver, pPort, hDMode）；
```

```
    Label1.Caption:=pDevice;
    Label2.Caption:=pDriver;
    Label3.Caption:=pPort;
end;
```

该例子先为存放设备名称、驱动文件名、端口号名而开辟空间，然后调用 GetProinter（）方法，并把返回的结果显示在标签上。通常情况下，驱动文件将返回一个空串。

Printer.SetPrinter（）方法的声明格式与 GetPrinter（）方法的声明格式完全一致，只是功能相反。

如果已经知道设备名、驱动文件名以及想要的端口名，那么可以调用 SetPrinter（）方法，如下所示：

```
Printer.SetPrinter（'HPLJ-IID',
                    'HPPCL',
                    'FILE',
                    0）;
```

该例子用于将打印文件输出到文件中。注意这里没有传送 DevMode 结构的句柄的参数。若有一个指向一个有效的 DevMode 结构的句柄，可以把它作为第四个参数使用。以后的章节将对该结构变量进行详细介绍。



跟我来

新建一个应用程序，然后放置一个 PrinterDialog 构件以及一个按钮构件到窗体上。清单 10-2 显示了如何将一串文本通过更改打印端口输出到文件中。

清单 10-2　获取打印机信息以及更改打印端口

```
unit Unit1;

interface
  uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
    StdCtrls;

  type
    TForm1 = class（TForm）
      Button1: TButton;
      PrintDialog1: TPrintDialog;
      Label1: TLabel;
      Label2: TLabel;
      Label3: TLabel;
```

```
    procedure Button1Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
uses Printers;

{$IFNDEF WIN32}
 const MAX_PATH = 144;
{$ENDIF}

procedure TForm1.Button1Click（Sender: TObject）;
var
  pDevice : pChar;
  pDriver : pChar;
  pPort   : pChar;
  hDMode : THandle;
  PDMode : PDEVMODE;
begin
  if PrintDialog1.Execute then begin
    GetMem（pDevice, cchDeviceName）;
    GetMem（pDriver, MAX_PATH）;
    GetMem（pPort, MAX_PATH）;
    //获得打印机信息，并将之显示在标签上
    Printer.GetPrinter（pDevice, pDriver, pPort, hDMode）;
    Label1.Caption:=pDevice;
    Label2.Caption:=pDriver;
    Label3.Caption:=pPort;
    //将打印内容输出至文件中
    Printer.SetPrinter（pDevice, PDriver, 'FILE:', hDMode）;
    FreeMem（pDevice, cchDeviceName）;
    FreeMem（pDriver, MAX_PATH）;
```

```
        FreeMem（pPort, MAX_PATH）;
        //开始打印
        Printer.BeginDoc;
        //向“打印机”输出文本
        Printer.Canvas.TextOut（100, 100, 'Delphi Is RAD!'）;
        //结束打印
        Printer.EndDoc;
    end;
end;

end.
```

这段代码中，先通过 GetPrinter（）方法获取打印机的设备名、驱动文件名以及端口名，并将这些信息输出到标签中，然后通过 SetPrinter（）方法来设置打印机的端口名，当设为“FILE”时表示将打印输出至文件中。最后，将一串字符串输出至文件中。

运行该程序，并单击按钮，将会弹出一个打印机设置对话框用以设置要使用的打印机，如果按了“确定”按钮，则标签将显示所选择的打印机的设备名、驱动文件名以及端口名，当运行语句

Printer.SetPrinter（pDevice, PDriver, 'FILE:', hDMode）;

时，又会弹出一个对话框，用以选择要输出到哪个文件中，文件的后缀名为.prn。最后会将一串字符串输出到所指定的文件中。



再送礼包

另外，还有一种方法可以实现永久的将打印输出至文件中。这种方法需要更改 Delphi 的源文件 printers.pas。

为了安全起见，先创建一个 patched 目录，并将 printers.pas 单元文件拷贝至新目录中，然后在拷贝的文件中进行更改。同时，应该将上述目录放在 library path 之首。重新启动 Delphi，然后在进行以下的更改后重新编译工程文件。注意，library path 可以在 Delphi 的“ToolslEnvironment optionsllibrary”菜单中找到。

下列步骤将实现上述目的：

（1）在 patched 目录下创建一个新单元文件 prnpage.pas，该文件中包含以下代码：

```
unit PrnPage;

interface

var FileNameToPrintTo : Pointer;

implementation
```

```
begin
  FileNameToPrintTo := Nil;
end.
```

（2）在 Patched 目录下打开 Printer.pas 文件，然后寻找“uses Consts;”并在 uses 语句中加入 prnpage 单元。

```
uses Consts, PrnPage;
```

（3）寻找“**lpszOutput := Nil;**”，并将之改成 lpszOutput := FileNameToPrintTo;

（4）将更改存盘，并重新启动 Delphi。

（5）为了测试所作的修改，输入以下代码。注意：当进行第一次编译时应确保使用“Build All”选项。

```
uses Printers, PrnPage;

{$IFNDEF WIN32}
 const MAX_PATH = 144;
{$ENDIF}

procedure TForm1.Button1Click（Sender: TObject）;
var
  lpPrnFile : pChar;
begin
  GetMem（lpPrnFile, MAX_PATH +1）;

  try
    StrPCopy（lpPrnFile, 'C:\Download\TestIt.Prn'）;
    FileNameToPrintTo := @lpPrnFile^;

    with Printer do begin
      BeginDoc;
      Canvas.TextOut（10, 10, 'Test'）;
      EndDoc;
    end;

    FileNameToPrintTo := Nil;

  finally
    FreeMem（lpPrnFile, MAX_PATH +1）;
  end;
end;
```

其功能与清单 10-2 类似，都是将字符串输出到文件中。



控制打印机设置



遇到难题

上一节讲述了通过 Printer.SetPrinter（）方法进行打印机端口的设置，但如果要进行对打印机的较深层次设置的话，该方法就显得无能为力了。例如，设置打印份数、指定打印质量或者设置纸张的来源等等。通常 Printer.SetPrinter（）方法的使用不值得推荐，因为大多数打印机必须由 Windows 安装才能工作。但在有些情况下这种方法还是有用的，例如，可以用此方法来改变一个给定的驱动文件将要使用的端口。

如何在代码中很好地控制打印机设置呢？



钥匙在此

为在代码中控制打印机设置，需要使用打印机的 TDeviceMode 结构。当前打印机的 TDeviceMode 结构可用 Printer.GetPrinter（）方法来获得。当获得 DevMode 结构的句柄时，就可用 Windows 的 GlobalLock（）函数来得到指向该结构的指针，然后可对该结构中不依赖设备的部分进行修改。

1. 使用 TDeviceMode 结构

每一个打印机都有一个唯一的 TDeviceMode 结构，TDeviceMode 结构的第一部分对所有打印机都相同，而第二部分则是专为驱动文件所用。可通过查询 TDeviceMode.dmSize 成员来找到驱动文件的 TDeviceMode 结构的公共部分的长度，同时也可以使用 TDeviceMode.dmDriverExtra 成员来得到驱动文件的 TDeviceMode 结构的私有部分的长度。若想复制和保存一个完整的 TDeviceModee 结构，则需要分配和复制（TDeviceMode.dmSize+TDeviceMode.dbDriverExtra）字节。

TdeviceMode 结构中包含了打印驱动程序的初始化信息和环境数据。程序员可以通过设置这个结构来得到或者设置打印机的属性，该结构在 windows.pas 中给予了定义。清单 10-3 列出了这个结构的声明。

清单 10-3　TDeviceMode 结构

```
TdeviceMode = packed record
  dmDeviceName: array[0..CCHDEVICENAME - 1] of AnsiChar;
```

```
    dmSpecVersion: Word;
    dmDriverVersion: Word;
    dmSize: Word;
    dmDriverExtra: Word;
    dmFields: DWORD;
    dmOrientation: SHORT;
    dmPaperSize: SHORT;
    dmPaperLength: SHORT;
    dmPaperWidth: SHORT;
    dmScale: SHORT;
    dmCopies: SHORT;
    dmDefaultSource: SHORT;
    dmPrintQuality: SHORT;
    dmColor: SHORT;
    dmDuplex: SHORT;
    dmYResolution: SHORT;
    dmTTOption: SHORT;
    dmCollate: SHORT;
    dmFormName: array[0..CCHFORMNAME - 1] of AnsiChar;
    dmLogPixels: Word;
    dmBitsPerPel: DWORD;
    dmPelsWidth: DWORD;
    dmPelsHeight: DWORD;
    dmDisplayFlags: DWORD;
    dmDisplayFrequency: DWORD;
    dmICMMethod: DWORD;
    dmICMIntent: DWORD;
    dmMediaType: DWORD;
    dmDitherType: DWORD;
    dmICCManufacturer: DWORD;
    dmICCModel: DWORD;
    dmPanningWidth: DWORD;
    dmPanningHeight: DWORD;
  end;
```

最基本的 TdeviceMode 结构定义成员有以下的公用成员：

```
    dmDeviceName: //设备名称
    dmSpecVersion://驱动程序所需的最低的 Windows 版本
    dmDriverVersion: //驱动文件的版本号
    dmSize: //TDeviceMode 的公用部分的长度
```

dmDriverExtra: //TDeviceMode 的私有部分的长度
dmFields: //TDeviceMode 标志位初始化部分
dmOrientation: //页的方向
dmPaperSize: //预定义的页大小或在定制大小时为零
dmPaperLength: //只在定制页时使用
dmPaperWidth: //只在定制页时使用
dmScale: //打印比例
dmCopies: //希望复制的份数
dmDefaultSource: //必须为零
dmPrintQuality: //预定义或打印机 X 方向的分辨率
dmColor: //若需要彩色打印时使用
dmDuplex: //若要双面打印时使用
dmYResolution: //零或者使用 dmPrintQuality 时在 x 方向的分辨率

在 TDeviceMode 的公用部分定义了一些额外的标志位，但并非适用于所有的打印机，除非确定所用的打印机支持这些额外的设置，否则最好不要使用它们。至于其他成员的含义和使用请参见 Delphi 的在线帮助。

为了得到当前打印机的 TDeviceMode 结构，可以首先调用 Printer.GetPrinter（）得到该结构在内存中的句柄，然后调用 GlobalLock（）得到指向该结构的指针。清单 10-4 显示了如何得到一个指向 TDeviceMode 结构的指针以及如何设置该结构中的成员。

清单 10-4　更改 TDeviceMode 结构中的成员

```
var
  pDevice, pDriver, pPort:array[0..255]of char;
  hDMode : THandle;
  PDMode : PDeviceMode;
begin
    //获得打印机的 DeviceMode 结构句柄
    Printer.GetPrinter（pDevice, pDriver, pPort, hDMode）;
    //如果句柄是 0，表示打印机没有装载
  if hDMode=0 then begin
    printer.PrinterIndex:=printer.PrinterIndex;
    Printer.GetPrinter（pDevice, pDriver, pPort, hDMode）;
  end;
  if hDMode<>0 then begin
    //根据 TDeviceMode 的句柄获得指向该结构的指针变量
    PDMode:=GlobalLock（hDMode）;
    if PDMode<>Nil then begin
      //设置 TDeviceMode 中的 dmCopies 成员，即设置打印的份数
      pdMode^.dmFields:=pdMode^.dmFields or DM_COPIES;
      pdMode^.dmCopies:=5;
```

```
            //解锁，释放内存
            GlobalUnlock（hDMode）;
            printer.PrinterIndex:=printer.PrinterIndex;
        end
    end;
    //开始打印
    Printer.BeginDoc;
    //向“打印机”输出文本
    Printer.Canvas.TextOut（100, 100, 'Delphi Is RAD!'）;
    //结束打印
    Printer.EndDoc;
end;
```

在该例中，定义了几个变量用于存放设备名、驱动文件名、端口名、以及指向 TDeviceMode 结构的句柄和指针。在该例中的第一行代码用于复位打印机。在调用了 GetPrinter 方法之后，检查指向该结构的一个有效句柄是否存在，如果存在，则锁住它并进行相应的修改，然后打开句柄锁，并复位打印机。应当注意的是，只有在开始一个打印任务之前才能进行这些修改。另外，还应该注意，并非所有的改变在所有的打印机上都支持。

2. 设置 TdeviceMode 成员

在清单 10-4 中已经讲述了如何设置 TDeviceMode 成员，从该例子中可以看到，为了设置这些成员，首先应设置 dmFields 域，以指定要设置的 TDeviceMode 成员，然后对要设置的域成员的值进行初始化。表 10-1 列出了 dmFields 域的各个位标志及其对应的成员。本部分将对 TDeviceMode 结构成员的设置进行详细的解释。

表 10-1　　dmFields 位标志

标 志	对应的字段
DM_ORIENTATION	dmOrientation
DM_PAPERSIZE	dmPaperSize
DM_PAPERLENGTH	dmPaperLength
DM_PAPERWIDTH	dmPaperWidth
DM_SCALE	dmScale
DM_COPIES	dmCopies
DM_DEFAULTSOURCE	dmDefaultSource
DM_PRINTQUALITY	dmPrintQuality
DM_COLOR	dmColor
DM_DUPLEX	dmDuplex
DM_YRESOLUTION	dmYResolution
DM_TTOPTION	dmTTOption
DM_COLLATE	dmCollate
DM_FORMNAME	dmFormName
DM_LOGPIXELS	dmLogPixels
DM_BITSPERPEL	dmBitsPerPel
DM_PELSWIDTH	dmPelsWidth
DM_PELSHEIGHT	dmPelsHeight
DM_DISPLAYFLAGS	dmDisplayFlags

续表

标 志	对应的字段
DM_DISPLAYFREQUENCY	dmDisplayFrequency
DM_ICMMETHOD	Windows 95 only: dmICMMethod
DM_ICMINTENT	Windows 95 only: dmICMIntent
DM_MEDIATYPE	Windows 95 only: dmMediaType
DM_DITHERTYPE	Windows 95 only: dmDitherType

（1）设置打印份数。可以通过设置 TDeviceMode 结构中的 copies 字段来告诉打印机要打印的份数。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_COPIES;
    dmCopies:=Copies;
end;
```

首先，要设置 dmFields 域的某个标志位，以指出 TDeviceMode 结构中哪个成员将被初始化。上例中，无论何时启动一个打印任务，所设的打印份数都将被传送给打印机。

（2）设置打印方向。可以通过设置 TDeviceMode 结构中的 dmOrientation 字段来告诉打印机的打印方向。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_ORIENTATION;
    dmOriention:= DMORIENT_PORTRAIT;
end;
```

该域成员可以设成 DMORIENT_PORTRAIT 或者 DMORIENT_LANDSCAPE 。

（3）设置纸张大小。可以通过设置 TDeviceMode 结构中的 dmPaperSize 字段来告诉打印机纸张的大小。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_PAPERSIZE;
    dmPaperSize:= DMPAPER_LETTER   ;//等等
end;
```

如果纸张的长度和宽度都已经通过 dmPaperLength 和 dmPaperWidth 成员设置完成的话，该域成员可以设成 0。否则，该域成员应该设置成下列定义的常量之一，其值见表 10-2。

表 10-2　　dmPaperSize 域成员值

值	含 义
DMPAPER_LETTER	Letter, 8 1/2- by 11-inches
DMPAPER_LEGAL	Legal, 8 1/2- by 14-inches
DMPAPER_A4	A4 Sheet, 210- by 297-millimeters
DMPAPER_CSHEET	C Sheet, 17- by 22-inches
DMPAPER_DSHEET	D Sheet, 22- by 34-inches
DMPAPER_ESHEET	E Sheet, 34- by 44-inches
DMPAPER_LETTERSMALL	Letter Small, 8 1/2- by 11-inches
DMPAPER_TABLOID	Tabloid, 11- by 17-inches
DMPAPER_LEDGER	Ledger, 17- by 11-inches

续表

值	含 义
DMPAPER_STATEMENT	Statement, 5 1/2- by 8 1/2-inches
DMPAPER_EXECUTIVE	Executive, 7 1/4- by 10 1/2-inches
DMPAPER_A3	A3 sheet, 297- by 420-millimeters
DMPAPER_A4SMALL	A4 small sheet, 210- by 297-millimeters
DMPAPER_A5	A5 sheet, 148- by 210-millimeters
DMPAPER_B4	B4 sheet, 250- by 354-millimeters
DMPAPER_B5	B5 sheet, 182- by 257-millimeter paper
DMPAPER_FOLIO Folio	8 1/2- by 13-inch paper
DMPAPER_QUARTO	Quarto, 215- by 275-millimeter paper
DMPAPER_10X14	10- by 14-inch sheet
DMPAPER_11X17	11- by 17-inch sheet
DMPAPER_NOTE	Note, 8 1/2- by 11-inches
DMPAPER_ENV_9	#9 Envelope, 3 7/8- by 8 7/8-inches
DMPAPER_ENV_10	#10 Envelope, 4 1/8- by 9 1/2-inches
DMPAPER_ENV_11	#11 Envelope, 4 1/2- by 10 3/8-inches
DMPAPER_ENV_12	#12 Envelope, 4 3/4- by 11-inches
DMPAPER_ENV_14	#14 Envelope, 5- by 11 1/2-inches
DMPAPER_ENV_DL	DL Envelope, 110- by 220-millimeters
DMPAPER_ENV_C5	C5 Envelope, 162- by 229-millimeters
DMPAPER_ENV_C3	C3 Envelope, 324- by 458-millimeters
DMPAPER_ENV_C4	C4 Envelope, 229- by 324-millimeters
DMPAPER_ENV_C6	C6 Envelope, 114- by 162-millimeters
DMPAPER_ENV_C65	C65 Envelope, 114- by 229-millimeters
DMPAPER_ENV_B4	B4 Envelope, 250- by 353-millimeters
DMPAPER_ENV_B5	B5 Envelope, 176- by 250-millimeters
DMPAPER_ENV_B6	B6 Envelope, 176- by 125-millimeters
DMPAPER_ENV_ITALY	Italy Envelope, 110- by 230-millimeters
DMPAPER_ENV_MONARCH	Monarch Envelope, 3 7/8- by 7 1/2-inches
DMPAPER_ENV_PERSONAL	6 3/4 Envelope, 3 5/8- by 6 1/2-inches
DMPAPER_FANFOLD_US	US Std Fanfold, 14 7/8- by 11-inches
DMPAPER_FANFOLD_STD_GERMAN	German Std Fanfold, 8 1/2- by 12-inches
DMPAPER_FANFOLD_LGL_GERMAN	German Legal Fanfold, 8 1/2- by 13-inches

（4）设置纸张长度。可以通过设置 TDeviceMode 结构中的 dmPaperLength 字段以 0.1 毫米的精度设置打印输出的长度。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_PAPERLENGTH;
    dmPaperLength:= PaperLength;
end;
```

对页长度的设置将取代对 dmPaperSize 域成员的设置。

（5）设置纸张宽度。可以通过设置 TDeviceMode 结构中的 dmPaperWidth 字段以 0.1 毫米的精度设置打印输出的宽度。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_PAPERWIDTH;
    dmPaperWidth:= PaperWidth;
```

```
end;
```

对页宽度的设置将取代对 dmPaperSize 域成员的设置。

（6）设置纸张打印比例。可以通过设置 TDeviceMode 结构中的 dmScale 域来设置打印的尺寸与实际尺寸的比值。因此，页的实际输出尺寸是物理尺寸与 dmscale 域值的乘积再除以 100。所以，如果打印输出（文本和图形）缩小为原始尺寸的一半，可以把 dmScale 设为 50，见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_SCALE;
    dmScale:= 50;
end;
```

（7）设置打印颜色。可以通过设置 TDeviceMode 结构中的 dmColor 域来指定输出单色还是彩色。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_COLOR;
    dmColor:= DMCOLOR_COLOR;//表示设置为彩色
end;
```

该域成员可以设置成的另一个值是 DMCOLOR_MONOCHROME，表示单色。

（8）设置打印质量。可以通过设置 TDeviceMode 结构中的 dmPrintQuality 域来指定打印的质量，即打印机输出的分辨率。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_PRINTQUALITY;
    dmPrintQuality:= DMRES_HIGH;//表示高分辨率打印
end;
```

该域成员还可以设置为其他三个值：DMRES_MEDIUM、DMRES_LOW 和 DMRES_DRAFT，分别表示中分辨率、低分辨率和草图质量打印。

（9）设置双面打印。可以通过设置 TDeviceMode 结构中的 dmDuplex 域来指定是否进行双面打印。设置方式见下例：

```
with PDMode^ do begin
    dmFields:=dmFields or DM_DUPLEX;
    dmDuplex:= DMDUP_SIMPLEX;
end;
```

该域成员还可以设置成其他两个值：DMDUP_HORIZONTAL 和 DMDUP_VERTICAL。



跟我来

如果现在准备打印一张相片，则一般需要进行高分辨率的打印，同时选择彩色打印（在打印机支持彩色打印的前提下）。新建一个应用程序，并在窗体上放置一个 OpenPictureDialog 控件、一个 Timage 控件以及一个按钮，下面的例子可以满足要求，清单 10-5 为该例子的完整代码。

清单 10-5　打印图片文件

```
unit printphoto;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtDlgs, ExtCtrls;

type
  TForm1 = class（TForm）
    Button1: TButton;
    OpenPictureDialog1: TOpenPictureDialog;
    Image1: TImage;
    procedure Button1Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
uses Printers;

procedure TForm1.Button1Click（Sender: TObject）;
var
  pDevice, pDriver, pPort:array[0..255]of char;
  hDMode : THandle;
  PDMode : PDeviceMode;
begin
   //获取要打印的图形文件
   if OpenPictureDialog1.Execute then begin
    //获得打印机的 DeviceMode 结构句柄
    Printer.GetPrinter（pDevice, pDriver, pPort, hDMode）;
    //如果句柄是 0，表示打印机没有装载
```

```
if hDMode=0 then begin
    printer.PrinterIndex:=printer.PrinterIndex;
    Printer.GetPrinter（pDevice, pDriver, pPort, hDMode）;
end;
if hDMode<>0 then begin
    //根据 TDeviceMode 的句柄获得指向该结构的指针变量
    PDMode:=GlobalLock（hDMode）;
    if PDMode<>Nil then begin
        //设置 TDeviceMode 中的 dmquality 成员为高分辨率打印
        pdMode^.dmFields:=pdMode^.dmFields or DM_PRINTQUALITY;
        pdMode^.dmPRINTQuality:=DMRES_HIGH;
        //设置 TDeviceMode 中的 dmcolor 成员为彩色打印
        pdMode^.dmFields:=pdMode^.dmFields or DM_COLOR;
        pdMode^.dmColor:=DMCOLOR_COLOR;
        //解锁，释放内存
        GlobalUnlock（hDMode）;
        printer.PrinterIndex:=printer.PrinterIndex;
      end
    end;
    //开始打印
    Printer.BeginDoc;
    image1.Picture.LoadFromFile（OpenPictureDialog1.FileName）;
    //将选中的图形文件输出到打印机中
    Printer.Canvas.Draw（10,10,image1.Picture.Graphic）;
    //结束打印
    Printer.EndDoc;
  end;
end;

end.
```



获取打印机详细信息



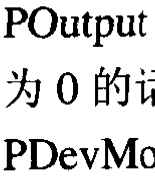
遇到难题

有些情况下需要得知打印机的详细信息，例如打印机是否支持某一特殊功能。典型的

例子就是打印机是否支持分段打印。分段打印是一种可以在一定的内存限制下提供打印速度和降低硬盘空间要求的技术。为了使用这种技术，必须调用 API 函数来设定此项功能。对于不支持这种技术的打印机来说，调用将不起作用。因此，首先确定打印机是否支持分段打印技术。如果不支持，则不需要调用这个 API 函数了。

又如，当不知道用户的绘图仪是否支持位图打印时，如果想使用绘图仪来输出位图的话，则应当先判断该绘图仪是否支持位图的输出方式。

像这些设备的信息是无法通过 Printer.GetPrinter（）方法来获取的，那么又该如何来获取该打印机的详细信息呢？



钥匙在此

为了获取打印机的详细信息，需要使用 Win32 API 函数 DeviceCapabilities（）来实现。实际上，该函数所获得的信息都来自于打印机的驱动程序。另外，支持 ANSI 字符和 Wide 字符的函数分别是 DeviceCapabilitiesA（）和 DeviceCapabilitiesW（）。

DeviceCapabilities（）函数在 WinPool.pas 单元文件中是这样进行声明的：

```
function DeviceCapabilities（pDevice, pPort: PChar; fwCapability: Word;
  pOutput: PChar; DevMode: PDeviceMode）: Integer; stdcall;
```

该函数用于获取一个打印机设备驱动程序的性能以及兼容性等信息。其中，参数 pDevice 为打印机的名称。注意，该参数为打印机的名称，而不是打印驱动程序的名称。

PPort 参数为与打印设备相连的端口名，例如"LPT1"等。

FwCapability 参数指定了要查询的打印机的性能，也就是本节要进行阐述的重要内容了。该参数可以取许多常量值（具体常量值参见 Win32 API 帮助文件的 DeviceCapabilities（）函数）。例如，当指定该参数为 DC_DUPLEX 时，表示要通过该函数来询问打印机的双面打印能力，当函数返回值为 1 时表示该打印机有双面打印的能力，否则，返回值为 0。

POutput 参数为一个字节数组。该数组的格式取决于 fwCapability 参数的设置。如果该参数为 0 的话，则该函数将返回需要输出数据的位数。

PDevMode 参数是指向一个 TDeviceMode 结构的指针。如果该参数为 Null 的话，则该函数将获得指定打印机驱动程序当前缺省的初始化值。否则，该函数将得到的值存放在 pDevMode 所指向的结构变量中。

如果函数执行成功，则返回值将取决于 fwCapability 参数的值。否则，返回值为-1。

如何从返回值中取出信息以及如何使用该函数及其参数，可以参见清单 10-6。该例子显示了如何获取打印机的进纸盒名称。

清单 10-6　获取打印机的进纸盒名称

```
procedure AddListViewItem（const aCaption, aValue: String; aLV: TListView）;
//该方法用于将标题和值加入 TListView 中
var
  NewItem: TListItem;
begin
```

```
    NewItem := aLV.Items.Add;
    NewItem.Caption := aCaption;
    NewItem.SubItems.Add（aValue）;
end;

procedure GetBinNames;
//该函数将所有可用的进纸盒位置名称加入到 TListView 中
var
    BinNames: Pointer;
    i: integer;
begin
{$R-} // 范围检测在此应当被关闭
// 首先确定有多少个进纸盒位置可用
Rslt := DeviceCapabilitiesA（Device, Port, DC_BINNAMES, Nil, Nil）;
if Rslt > 0 then
begin
  { 每一个进纸盒位置名称为 24 bytes 长，因此，
   分配 Rslt*24 个字节供装载进纸盒名}
  GetMem（BinNames, Rslt*24）;
  try
    // 现在可以通过分配的块获取进纸盒名
    if DeviceCapabilitiesA（Device, Port, DC_BINNAMES, BinNames, Nil） = -1 then
      raise Exception.Create（'DevCap Error'）;
    //将获得的信息加入到 list box 中
    AddListViewItem（'BIN NAMES', EmptyStr, lvGeneralData）;
    for i := 0 to Rslt - 1 do
    begin
      AddListViewItem（Format（'    Bin Name %d', [i]）,
          StrPas（TBinNames（BinNames^）[i]）, lvGeneralData）;
    end;
  finally
    FreeMem（BinNames, Rslt*24）;
  end;
 end;
{$R+} // 将范围检查打开
end;
```

说 明 关于参数 FwCapability 的内容较多，详细的介绍请参见 Win32 API 帮助文件中的 DeviceCapabilities（）函数说明。



跟我来

下面看一个获取打印机进纸盒名称的完整例子。新建一个应用程序，并在窗体上添加一个 Tcombobox 以及一个 TListView 控件，完整代码参见清单 10-7。

清单 10-7　获取打印机的进纸盒名称

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls;

type

  TForm1 = class（TForm）
    lvGeneralData: TListView;
    cbPrinters: TComboBox;
    procedure cbPrintersChange（Sender: TObject）;
    procedure FormCreate（Sender: TObject）;
  private
    { Private declarations }
    Device, Driver, Port: array[0..255] of char;
    ADevMode: THandle;
    procedure GetBinNames;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
uses
  Printers, WinSpool;
```

```
type
  // Types for holding bin names
  TBinName = array[0..23] of char;
  // Where used set $R- to prevent error
  TBinNames = array[0..0] of TBinName;

  var
  Rslt: Integer;


procedure AddListViewItem（const aCaption, aValue: String; aLV: TListView）;
//该方法用于将标题和值加入 TListView 中
var
  NewItem: TListItem;
begin
  NewItem := aLV.Items.Add;
  NewItem.Caption := aCaption;
  NewItem.SubItems.Add（aValue）;
end;

procedure TForm1.GetBinNames;
//该函数将所有可用的进纸盒位置名称加入到 TListView 中
var
  BinNames: Pointer;
  i: integer;
begin
{$R-} // 范围检测在此应当被关闭
  // 首先确定有多少个进纸盒位置可用
  Rslt := DeviceCapabilitiesA（Device, Port, DC_BINNAMES, Nil, Nil）;
  if Rslt > 0 then
  begin
    { 每一个进纸盒位置名称为 24 bytes 长，因此，
     分配 Rslt*24 个字节供装载进纸盒名}
    GetMem（BinNames, Rslt*24）;
    try
      // 现在可以通过分配的块获取进纸盒名
      if DeviceCapabilitiesA（Device, Port, DC_BINNAMES, BinNames, Nil） = -1 then
        raise Exception.Create（'DevCap Error'）;
      //将获得的信息加入到 list box 中
```

```
        AddListViewItem（'BIN NAMES', EmptyStr, lvGeneralData）;
        for i := 0 to Rslt - 1 do
        begin
          AddListViewItem（Format（'    Bin Name %d', [i]）,
            StrPas（TBinNames（BinNames^）[i]）, lvGeneralData）;
        end;
      finally
        FreeMem（BinNames, Rslt*24）;
      end;
    end;
{$R+} // 将范围检查打开
end;

procedure TForm1.cbPrintersChange（Sender: TObject）;
begin
  Screen.Cursor := crHourGlass;
  try
    // 选择使用的打印机
    Printer.PrinterIndex := cbPrinters.ItemIndex;
    with Printer do
      GetPrinter（Device, Driver, Port, ADevMode）;
    // 填充打印机的一般信息
    with lvGeneralData.Items do
    begin
      Clear;
      AddListViewItem（'Port', Port, lvGeneralData）;
      AddListViewItem（'Device', Device, lvGeneralData）;
      Rslt := DeviceCapabilitiesA（Device, Port, DC_DRIVER, Nil, Nil）;
      AddListViewItem（'Driver Version', IntToStr（Rslt）, lvGeneralData）;
    end;

    // 获取进纸盒名称，并将之显示到 Listview 中
    GetBinNames;
  finally
    Screen.Cursor := crDefault;
  end;
end;

procedure TForm1.FormCreate（Sender: TObject）;
```

```
begin
  //将所有可用的打印机加入到 combo box 中
  cbPrinters.Items.Assign（Printer.Printers）;
  //在 combo box 中显示缺省的打印机
  cbPrinters.ItemIndex := Printer.PrinterIndex;
  // 触发 the combo's OnChange 事件
  cbPrintersChange（Nil）;
end;

end.
```

该代码中，先在窗体创建时将当前缺省的打印机的名称添加到组合框中，然后引用该组合框的 OnChange（）事件。在组合框的 OnChange（）事件中，先将打印机的一般信息添加到 ListView 中，同时通过 DeviceCapabilitiesA（）函数取得打印机的驱动程序版本。然后，由函数 GetBinNames 将所有可用的进纸盒名称加入到 ListView 中。

运行该程序，得到结果如图 10-1 所示。由该结果可以看出，“我的电脑”上装载的打印机为 Epson LQ-100，端口为 LPT1，驱动程序的版本为 1025，进纸盒位置名称有自动送纸和手动送纸两种。

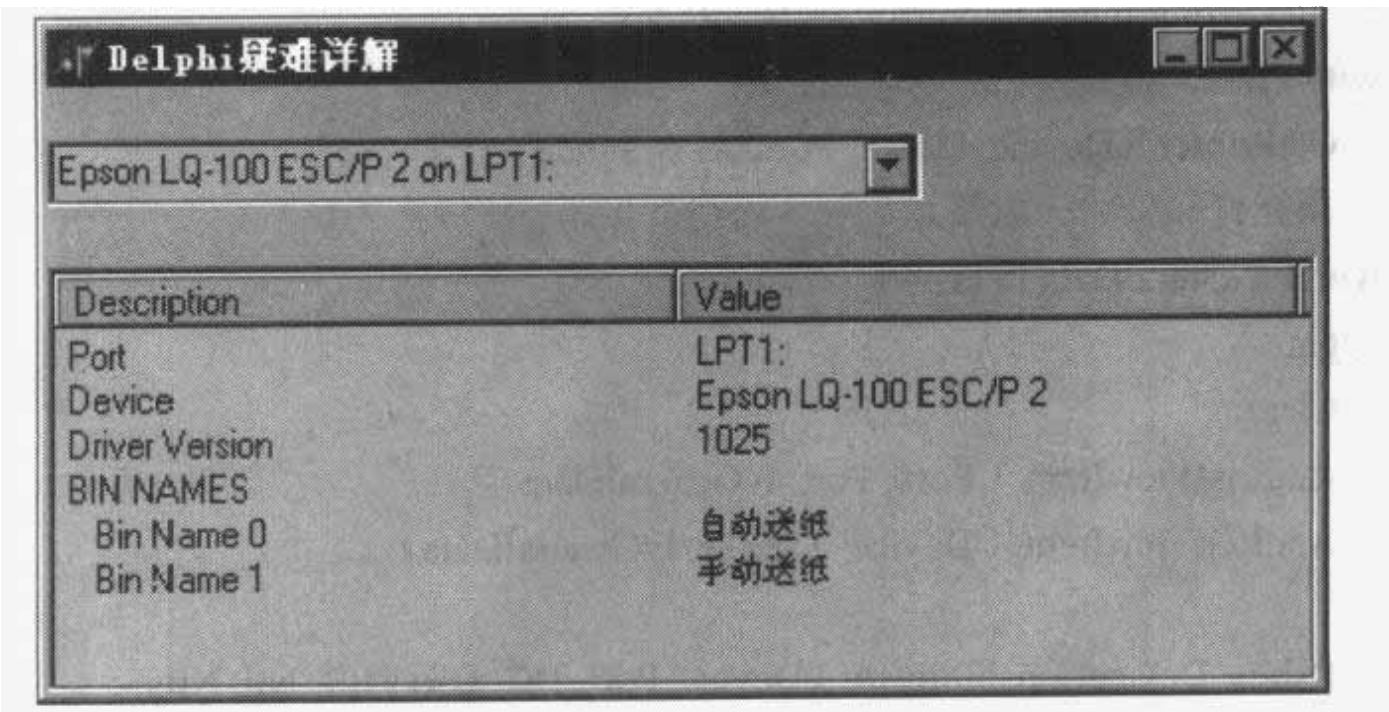


图 10-1　获取进纸盒名称



再送礼包

另外，还有几个函数可以获取打印机的详细信息，如 Win32 API 函数 GetDeviceCaps（）。该函数除了能获取打印机的设备信息之外，还可以获得绘图仪、显示器等设备的信息，而 DeviceCapabilities（）函数则只是获取打印机的信息。通常这些设备都有一个设备描述块，要获得有关设备的信息，可以向 GetDeviceCaps（）传递一个指向设备描述块的句柄，并指定所需要的索引号。

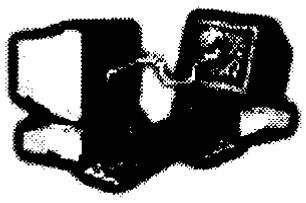
第十一章

Delphi 消息系统

热点透视

作为一个 Delphi 程序员，你也许已经发现，大部分情况下使用 VCL 的事件就已经足够了，只有少数情况下才需要深入到 Win32 的消息处理世界。不过，如果需要进行 Delphi 的更深层次编程，或者经常需要控制元件底层处理的话，就有必要对 Delphi 的消息系统有一个较深的理解。本章内容包括：

- 理解 Delphi 的消息系统
- 截获组件消息
- 自定义消息



理解 Delphi 的消息系统



遇到难题

作为一个 Delphi 程序员，你也许已经发现，大部分情况下使用 VCL 的事件就已经足够了，只有少数情况下才需要深入到 Win32 的消息处理世界。不过，如果需要进行 Delphi 的更深层次的编程，或者经常需要控制元件底层处理的话，就有必要对 Delphi 的消息系统有一个较深的理解。关于 Delphi 的消息系统也曾迷惑过许多 Delphi 程序员。



钥匙在此

1. Windows 消息系统

为了理解 Delphi 的消息系统，首先应当了解 Windows 消息系统是怎样工作的。Windows 的消息系统是由三个部分组成的：

- 消息队列。Windows 能够为所有的应用程序维护一个消息队列。应用程序必须从消息队列中获取消息，然后分派给某个窗口。
- 消息循环。应用程序从消息队列中检索消息，再把它分派给适当的窗口，然后继续从消息队列中检索下一条消息，再分配给适当的窗口，依此类推。
- 窗口过程。每个窗口都有一个窗口过程来接收传递给窗口的消息，它的任务就是获取消息，然后响应它。窗口过程实际上就是一个回调函数，处理了一个消息后，它通常要返回一个值给操作系统。

一个消息从发生到被一个窗口接收并响应，其中有 5 个步骤：

（1）系统发生了某个事件；

（2）Windows 把这个事件翻译成消息，然后把它放到一个消息队列中。

（3）应用程序从消息队列中接收到这个消息，并把它存放到一个消息记录中；

（4）应用程序把消息传递给一个适当的窗口，然后由这个窗口的窗口过程来处理；

（5）窗口过程对这个消息进行处理。

步骤（3）和（4）构成了应用程序的消息循环。消息循环往往是 Windows 应用程序的核心。因为消息循环使一个应用程序能够响应外部的事件。消息循环的任务就是从消息队列中检索消息，然后把消息传递给适当的窗口。如果消息队列中没有消息，Windows 就允许其他应用程序处理它们的消息。

2. Delphi 应用程序消息

在 Delphi 中由 Application.ProcessMessage（）从消息队列中查询消息，当该过程检索到该应用程序的消息后，将触发 Application.OnMessage（）事件。使用 OnMessage 可以截获发送到该应用程序的所有窗体的消息。该事件的声明格式如下：

```
type TMessageEvent = procedure （var Msg: TMsg; var Handled: Boolean） of object;
property OnMessage: TMessageEvent;
```

当应用程序收到一个窗体消息后，该事件就会发生。一个 OnMessage 事件句柄允许一个应用程序响应消息。如果应用程序对一个引入的消息没有一个特殊的处理过程的话，那么消息将被分发到相应的窗口中，Windows 将处理这个消息。该声明中，Msg 参数标识了 Windows 的消息，Handled 参数意味着事件的处理函数或者过程是否对此消息进行处理。如果该消息已经被完全处理了，为了防止后面继续处理该消息，应把此参数设置为 true。



OnMessage 仅仅当被 Post 到消息队列中的消息收到时才被激发，而不是那些使用 SendMessage（）函数发出的消息。

下例代码处理了一个定制的消息，当文件准备读取时由应用程序发送给自己。

```
const
  WM_FILEREADY = WM_USER + 2000;
procedure TForm1.FormCreate（Sender: TObject）;

begin
   Application.OnMessage := AppMessage;
end;

procedure TForm1.AppMessage（var Msg: TMsg; var Handled: Boolean）;
begin
   if Msg.message = WM_FILEREADY then
   begin
      Memo1.Lines.LoadFromFile（StrPas（PChar（Msg.lParam））） ;
      Handled := True;

   end;

   {对于其他消息应当将 Handled 设为 False }
   { 以便其他消息处理函数能被响应}
end;
```

3. Delphi 的 VCL 消息系统

所有的 Delphi 类有一个处理消息的内置机理，称之为消息处理方法或者 message handlers。Message handlers 的基本思想就是：类获取一些种类的消息，然后再分发（dispatch）它们，由响应的处理函数或者过程进行处理（调用哪一个处理函数或过程取决于所接受的消息）。对于一个特殊的消息，如果没有一个指定的方法存在的话，则由一个缺省的消息处理。

下图显示了消息分发系统：



VCL 定义了消息分发系统，该系统将所有的 Windows 消息（包括用户定义的消息）传递给相应的对象，由各对象的消息分发系统进行处理。

VCL 对象用于接收消息的方法叫做 MainWndProc（）。通过 MainWndProc（）可以对消息进行任何特殊的处理。不过，一般情况下很少直接调用 MainWndProc（）来处理消息，除非不想让消息通过 VCL 的消息系统分发。

从 MainWndProc（）返回后，消息被传递给对象的 WndProc（）方法，这就是该对象的窗体过程，然后进入 VCL 的分发机构。分发机构使用 Disptch（）方法把消息分发给一个消息句柄。

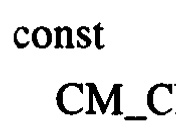
消息到达该消息的处理句柄（Handler）后，经过该句柄的处理，这个消息处理过程就结束了。

没有必要改变这个消息分发机理，所要做的只是产生一个消息处理方法（或者句柄，即一个消息处理函数）。

下例显示了如何自定义一个消息以及消息处理函数。

```
const
  CM_CHANGECOLOR = WM_APP + 400;
type
  TMyComponent = class（TControl）
  ...
protected
  procedure CMChangeColor（var Message: TMessage）; message CM_CHANGECOLOR;
end;

procedure TMyComponent.CMChangeColor（var Message: TMessage）;
begin
  Color := Message.lParam;
  inherited;
end;
```



跟我来

为了更好地理解 Delphi 的消息系统，下面创建一个小程序，分别用 Application.OnMessage 事件、WndProc（）、消息句柄和 DefaultHandler 来处理消息。完整代码见清单 11-1。

清单 11-1　消息处理

```
unit CIMain;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, Menus;

const
  SX_MYMESSAGE = WM_USER;                    // 自定义消息值
  MessString = '%s message now in %s.';   // String to alert user of message

type
  TMainForm = class（TForm）
    GroupBox1: TGroupBox;
    PostMessButton: TButton;
    WndProcCB: TCheckBox;
    MessProcCB: TCheckBox;
    DefHandCB: TCheckBox;
    SendMessButton: TButton;
    AppMsgCB: TCheckBox;
    EatMsgCB: TCheckBox;
    EatMsgGB: TGroupBox;
    OnMsgRB: TRadioButton;
    WndProcRB: TRadioButton;
    MsgProcRB: TRadioButton;
    DefHandlerRB: TRadioButton;
    procedure PostMessButtonClick（Sender: TObject）;
    procedure SendMessButtonClick（Sender: TObject）;
    procedure EatMsgCBClick（Sender: TObject）;
    procedure FormCreate（Sender: TObject）;
    procedure AppMsgCBClick（Sender: TObject）;
  private
    { 在应用程序层处理消息 }
    procedure OnAppMessage（var Msg: TMsg; var Handled: Boolean）;
    { 在窗体过程层处理消息}
    procedure WndProc（var Msg: TMessage）; override;
    { 在消息分发（dispatch）后处理消息 }
```

```
    procedure SXMyMessage（var Msg: TMessage）; message SX_MYMESSAGE;
    { 缺省的消息处理函数 }
    procedure DefaultHandler（var Msg）; override;
  end;

var
  MainForm: TMainForm;

implementation

{$R *.DFM}
const
  //用于显示消息是被 sent 还是被 posted
  SendPostStrings: array[0..1] of String = （'Sent', 'Posted'）;

procedure TMainForm.FormCreate（Sender: TObject）;
begin
  // 用 OnAppMessage 方法设置应用程序的 OnMessage 属性
  Application.OnMessage := OnAppMessage;
  // 使用 checkboxes 的 Tag 属性来存储与之相关的 radio 按钮的引用
  AppMsgCB.Tag := Longint（OnMsgRB）;
  WndProcCB.Tag := Longint（WndProcRB）;
  MessProcCB.Tag := Longint（MsgProcRB）;
  DefHandCB.Tag := Longint（DefHandlerRB）;
  //使用 radio 按钮的 Tag 属性来存储与之相关的 checkbox 的引用
  OnMsgRB.Tag := Longint（AppMsgCB）;
  WndProcRB.Tag := Longint（WndProcCB）;
  MsgProcRB.Tag := Longint（MessProcCB）;
  DefHandlerRB.Tag := Longint（DefHandCB）;
end;

procedure TMainForm.OnAppMessage（var Msg: TMsg; var Handled: Boolean）;
{ 应用程序的 OnMessage 处理函数 }
begin
  // 检查是否是用户自定义的消息
  if Msg.Message = SX_MYMESSAGE then
  begin
    if AppMsgCB.Checked then
    begin
```

```
        ShowMessage（Format（MessString, [SendPostStrings[Msg.WParam],
          'Application.OnMessage']））；
        Handled := OnMsgRB.Checked;
      end;
    end;
end;

procedure TMainForm.WndProc（var Msg: TMessage）；
{ Form 的窗口过程 WndProc 函数 }
var
  CallInherited: Boolean;
begin
  CallInherited := True;                    // 假设我们已经调用了继承的处理
  if Msg.Msg = SX_MYMESSAGE then     // 检查是否为用户自定义的消息
  begin
    if WndProcCB.Checked then
    begin
      ShowMessage（Format（MessString, [SendPostStrings[Msg.WParam], 'WndProc']））；
      CallInherited := not WndProcRB.Checked;
    end;
  end;
  if CallInherited then inherited WndProc（Msg）；
end;

procedure TMainForm.SXMyMessage（var Msg: TMessage）；
{ 用户自定义的消息处理函数 }
var
  CallInherited: Boolean;
begin
  CallInherited := True;
  if MessProcCB.Checked then
  begin
    ShowMessage（Format（MessString, [SendPostStrings[Msg.WParam],
      'Message Procedure']））；
    CallInherited := not MsgProcRB.Checked;
  end;
  if CallInherited then Inherited;
end;

procedure TMainForm.DefaultHandler（var Msg）；
```

```
{ 缺省的消息处理函数}
var
  CallInherited: Boolean;
begin
  CallInherited := True;
  if TMessage（Msg）.Msg = SX_MYMESSAGE then
  begin
    if DefHandCB.Checked then
    begin
      ShowMessage（Format（MessString, [SendPostStrings[TMessage（Msg）.WParam],
        'DefaultHandler']））;
      CallInherited := not DefHandlerRB.Checked;
    end;
  end;
  if CallInherited then inherited DefaultHandler（Msg）;
end;

procedure TMainForm.PostMessButtonClick（Sender: TObject）;
{ 向窗体 posts 消息}
begin
  PostMessage（Handle, SX_MYMESSAGE, 1, 0）;
end;

procedure TMainForm.SendMessButtonClick（Sender: TObject）;
{ 向窗体发送消息}
begin
  SendMessage（Handle, SX_MYMESSAGE, 0, 0）; // 向窗体发送消息
end;

procedure TMainForm.AppMsgCBClick（Sender: TObject）;
{ 当单击 checkbox 时使适当的 radio 按钮有效或者无效 }
begin
  if EatMsgCB.Checked then
  begin
    with TRadioButton（（Sender as TCheckBox）.Tag） do
    begin
      Enabled := TCheckbox（Sender）.Checked;
      if not Enabled then Checked := False;
    end;
  end;
```

```
end;

procedure TMainForm.EatMsgCBClick（Sender: TObject）;
{ 在适当时候使 radio 按钮有效或者无效}
var
  i: Integer;
  DoEnable, EatEnabled: Boolean;
begin
  // 获取有效/无效标志
  EatEnabled := EatMsgCB.Checked;
   for i := 0 to EatMsgGB.ControlCount - 1 do
    with EatMsgGB.Controls[i] as TRadioButton do
      begin
        DoEnable := EatEnabled;
        if DoEnable then DoEnable := TCheckbox（Tag）.Checked;
        if not DoEnable then Checked := False;
        Enabled := DoEnable;
      end;
  end;

  end.
```

在该代码中，当窗体创建时用 OnAppMessage 方法设置应用程序的 OnMessage 属性，以便当应用程序获取消息（Posted 消息）后，可以触发该事件。

在该代码中，包含了 PostMessButton 和 SendMessButton 的 OnClick 事件。前者调用 PostMessage（）把一个自定义的消息发送给窗体，后者调用 SendMessage（）把一个自定义消息发送给窗体。注意，为了区分两个不同的发送消息方式，这两个函数传递的参数不一样，前者的 Wparam 参数值设为 1，后者的 Wparam 参数值为 0。

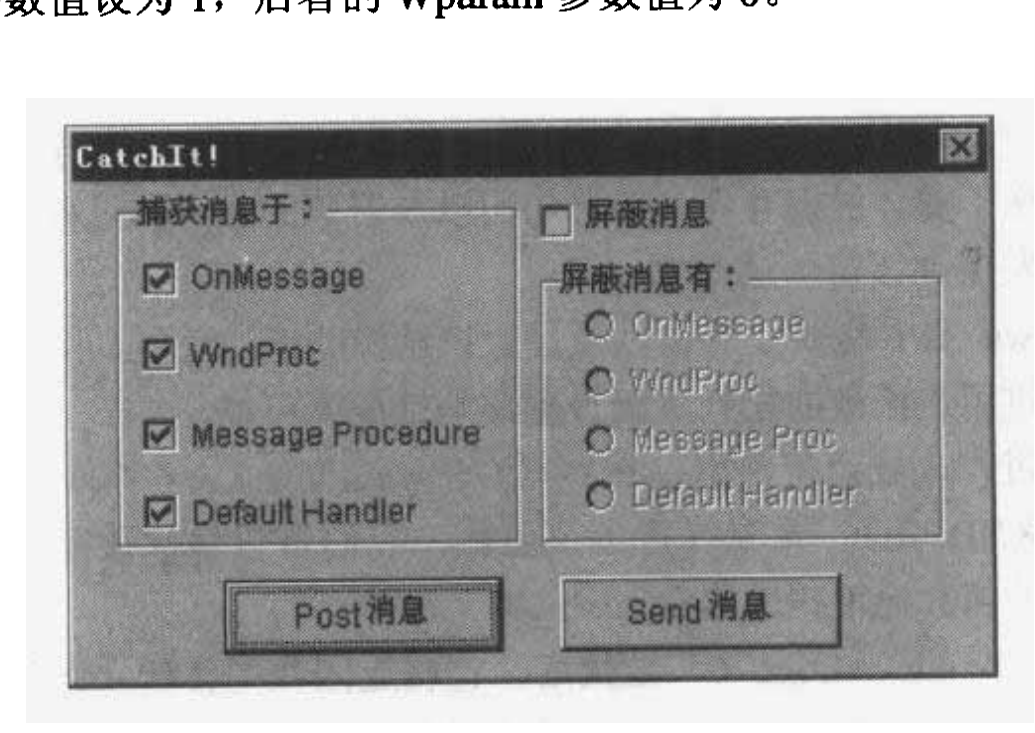


图 11-1　消息系统

运行该程序，结果如图 11-1 所示。如果选择了左列所有的 checkbox，当单击 PostMessButton 按钮时，先后会经过 OnMessage、WndProc、Message 以及 DefaultHandler 等四个过程；当单击 SendMessButton 按钮时，只会经过后三个过程。这一方面表明消息经过处理的先后顺序，同时也表明了当程序中使用 SendMessage（）时，应用程序的 OnMessage 事件不会发生。在程序主界面的右列为用来屏蔽一些消息的选项。



截获组件消息



遇到难题

Delphi 的 VCL 已经对大量控件或者窗口组件进行了很好的封装，这些 VCL 构件如此方便地使用，作为程序员肯定体会比较深刻。但是，物极必反，虽然 Delphi 已经很好地将这些消息封装成了事件，但对于这些事件并未全部覆盖该控件应该进行处理的消息，尽管 Delphi 已经对其进行了缺省处理，但有时程序员需要截获一些消息，而事件过程并未提供的话，此时就需要进行编程来处理了。



钥匙在此

对于如何截获消息，其实在前一章节中就已经有所讲述了。但本节将用另外一种方法来实现消息的截获。

Delphi 提供了许多方法来为 VCL 控件截获消息，你也许希望有一个快速而有效的方法来“偷”和替代一个已经给定的窗口过程。通过这种方法就可以有效地为任何一个窗体或者有 Window 句柄属性的 VCL 控件截获所有的窗体消息。

对于创建的每一个窗体而言，系统将创建一个载有关于窗口的消息结构。在这个结构中所载有的信息包括了窗体的拥有者、信息句柄以及窗口的主窗体过程。所有关于该窗体的 Windows 消息都被发送到该窗体过程中。

微软的 Windows API 提供了获取以及设置该结构的值的功能，从而使截取一个窗口主过程的地址以及通过用户函数的指针重新设置该地址成为可能。

为了替代窗口过程，我们将使用 API 函数 SetWindowsLong（），并且将我们要操作的窗口句柄、GWL_WNDPROC 常量（告诉该函数，我们希望替代该窗口过程）以及我们希望用来替代它自己的函数地址传递给它。

当我们调用 SetWindowsLong（）函数后，它将返回我们替代它之前的值。同时，我们也要记住该值，以便我们可以调用先前的过程来处理所有的我们并不关心的其余消息。当我

们处理完后，应该重新安装这个过程。以便使消息的处理能像以往一样返回正常状态。

下面将详细讲述有关的几个 API 函数。

1. GetWindowLong（）函数

SetWindowLong（）函数在 Windows.pas 中的声明如下：

function SetWindowLong（hWnd: HWND; nIndex: Integer;

dwNewLong: Longint）: Longint; stdcall;

SetWindowLong（）函数用于改变指定窗口的属性。该函数也用于设置 32 位值在窗口额外的内存中指定的偏移量。其中参数 hWnd 为窗口或者窗体所属类的句柄。

参数 nIndex 指定要设置的值基于 0 的偏移量，有效的值在从 0 到额外内存的字节数减 4 的范围内。例如，设置的额外内存为 12 个字节，则 8 应该是第三个 32 位整数的索引值。为了设置任何一个其他值，可以指定为下列值之一，如表 11-1 所示。

表 11-1　　nIndex 参数值

值	含 义
GWL_EXSTYLE	设置一个新扩展的窗体类型
GWL_STYLE	设置一个新的窗体类型
GWL_WNDPROC	设置一个新的窗体过程的地址
GWL_HINSTANCE	设置一个新应用实例句柄
GWL_ID	设置一个窗体的标识符
GWL_USERDATA	设置与窗体关联的 32 位值。每一个窗体有一个相应的 32 位值，该值被创建窗体的应用程序所使用

当 hwnd 参数表示一个对话框时，下列值也是可用的。

DWL_DLGPROC　　设置对话框函数的新地址。

DWL_MSGRESULT　　在对话框过程中设置要处理消息的返回值。

DWL_USER　　设置一个新的额外信息，这些信息对应用程序来说是私有信息，例如句柄或者指针。

参数 dwNewLong 指定要替换的值。

如果函数运行成功，则返回值即为先前指定的 32 位整数值。如果失败，则返回值为 0。

如果使用 SetWindowLong（）函数以及 GWL_WNDPROC 索引可以替换窗体过程。

2. CallWindowProc（）函数

该函数将消息信息传递给指定的窗体过程。它在 windows.pas 文件中的声明格式如下：

function CallWindowProc（lpPrevWndFunc: TFNWndProc; hWnd: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM）: LRESULT; stdcall;

参数 pPrevWndFunc 表示先前过程的指针。参数 hWnd 为获取消息窗体的句柄。Msg 用于指定消息。

wParam、lParam 指定附加的特殊消息的信息。这个参数的内容取决于 Msg 参数的值。

返回值表示处理消息的结果。

3. WindowProc（）函数

如何在新定义的窗口过程中处理消息呢?

WindowProc（）函数是应用程序定义的回调函数，该函数用于处理被发送到窗体的消息。窗口过程的定义必须满足一定格式，其在 API 中的声明格式为：

```
LRESULT CALLBACK WindowProc（
    HWND hwnd, // 窗口的句柄
    UINT uMsg,  // 消息标识符，即消息常量
    WPARAM wParam,     // 第一个消息参数
    LPARAM lParam   //第二个消息参数
  ）;
```

消息处理的结果即返回的值，具体返回值的含义取决于被发送的消息的种类。

为了定义新的窗体过程，在其后不要忘了加上一个调用协议，对于 32 位系统来说，应该指定调用协议为 stdcall，因为该函数是被系统一级调用的。

例如，下述语句新定义了一个 WindowProc 函数：

```
function NewWindowProc（WindowHandle : hWnd;
                         TheMessage    : WParameter;
                         ParamW        : WParameter;
                         ParamL        : LParameter）  : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
{ 在这里处理您自己的消息 }
  canse TheMessage  of
    …
  end;
{ 如果要停止处理其他更多的消息的话，在这里退出并且返回 0}
{ 调用老窗口过程以允许对消息的缺省处理}
  NewWindowProc := CallWindowProc（OldWindowProc,
                                   WindowHandle,
                                   TheMessage,
                                   ParamW,
                                   ParamL）;
  end;
```

WindowProc 过程对于 VCL 控件而言，已经封装成为一个参数，VCL 控件的声明格式为：

```
type TWndMethod = procedure（var Message: TMessage）  of object;
property WindowProc: TWndMethod;
```



跟我来

新建一个应用程序，然后在窗体上放置一个 Table、TDBGrid 以及 TDataSource 控件，并设置其相应的属性，使 table 与数据库表相连接并打开表。在例子代码中，我们将截获 TDBGrid 组件的 WM_VSCROLL 消息，该消息提示我们用户已经滚动了垂直方向的滚动条。既然该控件或者窗口的所有消息都首先通知给我们的函数，那么在该消息被构件收到之前，我们就能够有效地截获和处理任何一个事件。完整代码如清单 11-2 所示。

其他您关心的要捕获的通知消息可以在 Win32 SDK 的帮助系统中查找到，它们一般以 WM_打头。

注意，以下代码可以在 16 位或者 32 位系统中进行编译使用。

清单 11-2　截获控件消息

```
unit WinProc1;
interface

uses
{$IFDEF WIN32}
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, DB, DBTables;
{$ELSE}
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, DB, DBTables, Grids, DBGrids;
{$ENDIF}

type
  TForm1 = class（TForm）
    DBGrid1: TDBGrid;
    Table1: TTable;
    DataSource1: TDataSource;
    procedure FormCreate（Sender: TObject）;
    procedure FormDestroy（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

```
  Form1: TForm1;

implementation

{$R *.DFM}

type
{$IFDEF WIN32}
  WParameter = LongInt;
{$ELSE}
  WParameter = Word;
{$ENDIF}
  LParameter = LongInt;

{声明一个变量用于存放我们要替代的窗口过程  }
var
  OldWindowProc : Pointer;

function NewWindowProc（WindowHandle : hWnd;
                         TheMessage    : WParameter;
                         ParamW        : WParameter;
                         ParamL        : LParameter）  : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
{ 在这里处理您自己的消息 }
  if TheMessage = WM_VSCROLL then begin
    ShowMessage（'The vertical scrollbar is scrolling!'）;
  end;
  { 如果要停止处理其他更多的消息的话，在这里退出并且返回 0}
  { 调用老窗口过程以允许对消息的缺省处理，并将处理的消息结果
  返回给新的窗体过程}
  NewWindowProc := CallWindowProc（OldWindowProc,
                                   WindowHandle,
                                   TheMessage,
                                   ParamW,
                                   ParamL）;
end;

procedure TForm1.FormCreate（Sender: TObject）;
begin
```

```
{ 为控件指定新的窗口过程，并且记住老的窗口过程的地址}
  OldWindowProc := Pointer（SetWindowLong（DbGrid1.Handle,
                                          GWL_WNDPROC,
                                          LongInt（@NewWindowProc））；
end;

procedure TForm1.FormDestroy（Sender: TObject）；
begin
{ 重新设置为老的窗口过程 }
  SetWindowLong（DbGrid1.Handle,
                 GWL_WNDPROC,
                 LongInt（OldWindowProc））；
end;

end.
```

运行该应用程序，如果滚动 DBGrid 的滚动条时，应用程序将给以相应的提示。



自定义消息



遇到难题

在某些情况下，一个应用程序可能需要向自己发送消息，或者在两个应用程序之间发送消息。这时候就会产生一个问题：“为什么要发送消息而不是直接调用一个过程呢？”。这个问题问得好，它有这么几个答案。首先，消息使您不需要知道接收者的确切类型。其次，消息可以有选择地进行处理，如果接收者对消息没有作任何处理，则不会造成任何后果。最后，消息可以广播给多个接收者，而如果要同时调用几个过程则比较困难。



钥匙在此

对于自定义消息，下面将分几种情况进行讨论。

1. 在同一个应用程序内部发送消息

对于自定义消息，一般包括了声明消息常量、声明和定义一个消息处理过程以及发送或激发消息这三部曲。

声明消息常量，例如：

```
const
  myMessage=WM_USER+100;
```

声明和定义一个消息处理过程或者句柄，例如：

```
…
procedure myMessageHandle（var msg:TMessage）;message mymessage;
…
procedure TForm1.myMessageHandle（var msg:TMessage）;
begin
    showmessage（'自定义消息！'+
                #13+'参数 1：'+inttostr（msg.WParam）+
                #13+'参数 2：'+inttostr（msg.LParam）
                ）;
end;
```

发送消息可以使用三种方式，一种是使用 PerForm（）函数，该函数的声明格式是：

```
function TControl. Perform（Msg: Cardinal; WParam, LParam: Longint）: Longint;
```

另一种是使用 API 函数 SendMessage（），该函数的声明格式是：

```
function SendMessage（hWnd: HWND; Msg: UINT; wParam: WPARAM;
lParam: LPARAM）: LRESULT; stdcall;
```

第三种是使用 API 函数 PostMessage（），该函数的声明格式是：

```
function PostMessage（hWnd: HWND; Msg: UINT; wParam: WPARAM;
lParam: LPARAM）: BOOL; stdcall。
```

例如，这些函数的调用方式如下述代码所示：

```
    Form1.PerForm（mymessage,0,0）;
    postmessage（Form1.handle,mymessage,1,1）;
    sendmessage（Form1.handle,mymessage,2,2）;
```

2. 在应用程序之间发送消息

如果要在两个或者多个应用程序之间发送消息，则每个应用程序最好要调用 RegisterWindowMessage（）函数，以保证整个系统中某个消息的编号是一致的。

```
function RegisterWindowMessage（lpString: PChar）: UINT; stdcall;
```

该函数需要传递一个字符串，并返回一个新的消息常量（$C000~$FFFF）。为了互相能够接收和发送该消息，每一个应用程序都应当使用相同的字符串，而 Windows 也会返回相同的消息常量值。调用该函数的好处是，使用不同的消息字符串时，可以返回不同的消息常量。这样，您就可以放心地向所有的窗口广播同一个消息了。

清单 11-3 为两个程序的片断，说明了应用程序间如何定义和发送消息。

清单 11-3　应用程序间发送消息

```
{*********************************}
{          程 序 一               }
{*********************************}
//要发送消息的程序片段
procedure TForm1.Button2Click（Sender: TObject）;
const
    MyMessageName='jxg';
var
    MyMsg: Cardinal;
begin
  //在系统中登记想要发送的消息名(其实只是一个名称而已),获得该系统消息的 ID 号
  MyMsg := RegisterWindowMessage（MyMessageName）;
  //当登记成功后，向另一个 Window 或应用程序发送所登记的消息
  if MyMsg<>0 then
      sendmessage（findwindow（Nil,'Form2'）,mymsg,1973,0509）;//
end;

{*********************************}
{          程 序 二               }
{*********************************}
//接收消息的程序片段
//该程序的 Caption='Form2'
procedure   Tform2.wndproc（var message:TMessage）;
const
    MyMessageName='jxg';
var
    MyMsg: Cardinal;
begin
    //获取要处理的消息的 ID 号
    MyMsg:=RegisterWindowMessage（MyMessageName）;
//如果 message.Msg=所登记的消息 ID
    //表示所接收的消息 message 正是要处理的消息
    //进行相应的处理
    if message.Msg=myMsg then begin
        showmessage（'我已经收到你发的消息，名为'+MyMessageName+#13+
                    '参数 1='+inttostr（Message.Wparam）+#13+ //1973
                    '参数 2='+inttostr（message.lParam））;      //509
    end;
```

```
    inherited WndProc（message）;
end;
```



跟我来

下面列举了一个在应用程序内部进行消息发送以及在应用程序之间发送消息的例子。完整代码参见清单 11-4。

清单 11-4　自定义消息并发送

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

const
  myMessage=WM_USER+100;
  mymessageName='jxg_message';

type
  TForm1 = class（TForm）
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    procedure Button1Click（Sender: TObject）;
    procedure Button2Click（Sender: TObject）;
    procedure Button3Click（Sender: TObject）;
    procedure Button4Click（Sender: TObject）;
  private
    { Private declarations }
    procedure myMessageHandle（var msg:TMessage）;message mymessage;
    procedure    wndproc（var message:TMessage）;override;

  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.myMessageHandle（var msg:TMessage）;
begin
    showmessage（'自定义消息！'+
                #13+'参数 1：'+inttostr（msg.WParam）+
                #13+'参数 2：'+inttostr（msg.LParam）
                ）;
end;

procedure TForm1.Button1Click（Sender: TObject）;
begin
     Form1.PerForm（mymessage,0,0）;
end;

procedure TForm1.Button2Click（Sender: TObject）;
begin
    postmessage（handle,mymessage,1,1）;
end;

procedure TForm1.Button3Click（Sender: TObject）;
begin
     sendmessage（handle,mymessage,2,2）;
end;

procedure  TForm1.wndproc（var message:TMessage）;
var
   MyMsg: Cardinal;
begin
    //获取要处理的消息的 ID 号
    MyMsg:=RegisterWindowMessage（MyMessageName）;

    //如果 message.Msg=所登记的消息 ID，表示所接收的消息 message 正是要处理的消
息
```

```
    //进行相应的处理
    if message.Msg=myMsg then begin
        showmessage（'我已经收到你发的消息，名为'+MyMessageName+#13+
                    '参数 1='+inttostr（Message.Wparam）+#13+ //1973
                    '参数 2='+inttostr（message.lParam））；      //509
    end;

    inherited WndProc（message）；
end;

procedure TForm1.Button4Click（Sender: TObject）；
var
    MyMsg: Cardinal;
begin
    //获取要处理的消息的 ID 号
    MyMsg:=RegisterWindowMessage（MyMessageName）；
    //向所有的窗体广播消息
    sendmessage（HWND_BROADCAST,myMsg,4,4）；
end;

end.
```

在该代码中，由 Button4 的 click 事件向所有窗体发送消息，这是由参数 HWND_BROADCAST 所指定的。本程序在运行时同样是所有窗体中的一个窗体，所以，当该按钮发生单击事件时，本程序的 Form1 的窗体过程也会接收到该消息，并会给以相应的提示。

第十二章

Windows 系统控制

热点透视

要实现对 Windows 系统的控制，对于封装好的控件来说已经无能为力了，因此，需要直接调用 Windows API 函数才可能达到目的。本章内容包括：

- ✧ 创建控制面板小应用程序
- ✧ 调用控制面板设置功能
- ✧ 检测即插即用硬件发生的变化
- ✧ 直接控制 I/O
- ✧ 获取系统一般信息
- ✧ 检查环境变量
- ✧ 设置桌面墙纸
- ✧ 捕获系统按键



创建控制面板小应用程序



遇到难题

控制面板应用程序是一个具有特殊目的的动态连接库，用于让用户配置操作系统的环境。虽然系统中提供了很多标准的控制面板应用程序，但您也可以创建另外的控制面板应用程序，从而使用户可以检查和修改指定硬件或者软件的设置和操作模式。



钥匙在此

控制面板最基本的任务就是显示一个对话框，并且执行一些指定的任务。

控制面板应用程序一般被指定的操作系统工具所控制，用户可以访问这些应用程序。然而，在一些应用程序中也可以装载和管理控制面板应用程序，只要这些进行控制的应用程序向控制面板应用程序发送消息和处理控制面板应用程序所返回的值即可。

大部分控制面板应用程序显示和管理单个对话框，它提供给用户进行一些设置。然而，任何一个给定的控制面板应用程序都可以提供很多的对话框来控制一些系统组件。这些单独的对话框一般称之为 applets。为了区别对话框，控制面板应用程序为每一个对话框提供了独一无二的图标。控制应用程序显示这些图标，用户可以通过选择相应的图标来选择对话框。

每一个控制面板应用程序必须输出一个标准的接口函数 CplApplet。这个函数获得请求后，便形成控制面板（CPL）消息，并且执行所请求的任务，例如初始化应用程序、显示和管理对话框以及关闭应用程序等。

当控制应用程序首次装载控制面板应用程序时，它可以获得 CplApplet 函数的地址并随后使用该地址调用函数并传递消息。控制应用程序可以发送后面所述的控制面板消息。

简单地创建一个以.cpl 结尾的动态连接库（Control Panel Library）文件，并且将之放到 Windows 的系统目录（system）下。每一个 CPL 文件可以支持多个控制面板应用程序（applets）。CPL 中有一个函数接口叫做 CPlApplet（），该函数必须置入 export 指令中作为调用的入口。所有下列这些控制面板消息将随着单一入口一起被提供给控制应用程序。

CPL_INIT 被发送，并指示 CPlApplet（） 被发现。返回值为 True 则继续装载的处理。

CPL_GETCOUNT 返回被.cpl 所支持的 applet 的个数。

CPL_INQUIRE 被发送，用于获取关于每一个 applet 所支持的信息。lParam1 载有基于 0 的所使用 applet 的个数。lParam2 为 TcplInfo 结构类型指针，该结构中 idIcon 域应该使用

显式的资源 ID 号进行初始化，idName 和 idInfo 域应该使用资源字符串 ID 初始化名称和描述。Idata 载有 applet 所定义的数据。

CPL_SELECT 当 applet 的图标被用户选择时被发送。lParam1 载有被选中的 applet 数目。lParam2 载有 applet 用户定义的数据值。

CPL_DBLCLK 当 applet 图标被双击时被发送。lParam1 载有被选中的 applet 个数。lParam2 载有 applet 用户定义的数据。这个消息初始化 applet 对话框的显示。

CPL_STOP 当退出控制面板时向每一个 applet 发送。lParam1 载有 applet 个数。lParam2 载有 applet 用户定义的数据。这个消息初始化 applet 对话框的显示。当进行这个调用时任何一个 applet 细节都被清除。

CPL_EXIT 先于控制面板调用 FreeLibray 时被发送。在进行这个调用时，没有 applet 被清除。

CPL_NEWINQUIRE 与 CPL_INQUIRE 相类似，只不过参数 lParam2 表示 NEWCPLINFO 结构变量指针。

您的控制面板将需要附加一些资源给该函数。您将需要创建载有控制面板 applet 的名称和描述信息以及图标的资源文件（资源文件如何创建参见相关的章节）。

例如，.rc 文件中载有带有两个字符串以及一个图标指针的字符串表：

```
STRINGTABLE
{
 1, "TestApplet"
 2, "My Test Applet"
}

2 ICON C:\ Borland\Borland Shared\Images\Icons\handshak.ICO
```

为了将这个.rc 文件编译成能够连接到您的应用程序中的 res 文件，在 DOS 命令行中敲入以下语句即可，例如：

```
c:\Delphi\Bin\brcc32.exe c:\Delphi\MyRes.rc
```

当编译完成之后，在您的.rc 文件所在目录下已经多了一个.res 文件,这就是要使用的经过编译的资源文件。如果您需要在多平台下开发应用程序的话，则应该创建连接到您应用程序中的 16 位以及 32 位的资源文件。



跟我来

下面是一个控制面板 applet 的例子，该 applet 在响应 CPL_DBLCLK 消息时执行第二个可执行程序。您也可以调整代码来显示一个窗体或者一个对话框。该例子可以应用于 16 位或者 32 位操作系统中。

为创建工程，您需要将上述的.rc 文件编译成一个资源文件，命名成 TCPL32.RES 或者 TCPL16.RES。完整例子参见清单 12-1。

清单 12-1　创建控制面板 applet

```
library TestCpl;

{$IFDEF WIN32}
uses
   SysUtils,
   Windows,
   Messages;
{$ELSE}
uses
   SysUtils,
   WinTypes,
   WinProcs,
   Messages;
{$ENDIF}

{$IFDEF WIN32}
 {$R TCPL32.RES}
{$ELSE}
 {$R TCPL16.RES}
{$ENDIF}

const NUM_APPLETS = 1;

{$IFDEF WIN32}
 const CPL_DYNAMIC_RES = 0;
{$ENDIF}
const CPL_INIT = 1;
const CPL_GETCOUNT = 2;
const CPL_INQUIRE = 3;
const CPL_SELECT = 4;
const CPL_DBLCLK = 5;
const CPL_STOP = 6;
const CPL_EXIT = 7;
const CPL_NEWINQUIRE = 8;
{$IFDEF WIN32}
 const CPL_STARTWPARMS = 9;
{$ENDIF}
const CPL_SETUP = 200;
```

```
{$IFNDEF WIN32}
type DWORD = LongInt;
{$ENDIF}

type TCplInfo = record
        idIcon : integer;
        idName : integer;
        idInfo : integer;
        lData : LongInt;
      end;
      PCplInfo = ^TCplInfo;

type TNewCplInfoA = record
        dwSize : DWORD;
        dwFlags : DWORD;
        dwHelpContext : DWORD;
        lData : LongInt;
        IconH : HIcon;
        szName : array [0..31] of char;
        szInfo : array [0..63] of char;
        szHelpFile : array [0..127] of char;
      end;
      PNewCplInfoA = ^TNewCplInfoA;

{$IFDEF WIN32}
type TNewCplInfoW = record
        dwSize : DWORD;
        dwFlags : DWORD;
        dwHelpContext : DWORD;
        lData : LongInt;
        IconH : HIcon;
        szName : array [0..31] of WChar;
        szInfo : array [0..63] of WChar;
        szHelpFile : array [0..127] of WChar;
      end;
      PNewCplInfoW = ^TNewCplInfoW;
{$ENDIF}

type TNewCplInfo = TNewCplInfoA;
```

```
type PNewCplInfo = ^TNewCplInfoA;

function CPlApplet（hWndCPL : hWnd;
                    iMEssage : integer;
                    lParam1 : longint;
                    lParam2 : longint）  : LongInt
{$IFDEF WIN32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
  case iMessage of
    CPL_INIT : begin
      Result := 1;
      exit;
    end;
    CPL_GetCount : begin
      Result := NUM_APPLETS;
      exit;
    end;
    CPL_Inquire : begin

      PCplInfo（lParam2）^.idIcon := 2;
      PCplInfo（lParam2）^.idName := 1;
      PCplInfo（lParam2）^.idInfo := 2;
      PCplInfo（lParam2）^.lData := 0;
      Result := 1;
      exit;
    end;
    CPL_NewInquire : begin
      //当控制面板管理程序刷新显示时发送该消息
      //填写 TNewCplInfo 结构信息
      PNewCplInfo（lParam2）^.dwSize := sizeof（TNewCplInfo）;
      PNewCplInfo（lParam2）^.dwHelpContext := 0;
      PNewCplInfo（lParam2）^.lData := 0;
      PNewCplInfo（lParam2）^.IconH := LoadIcon（hInstance,
                                          MakeIntResource（2））;
      //控制面板中显示的图标
      //从资源文件中通过指定图标的 ID 号获取图标
      lStrCpy（@PNewCplInfo（lParam2）^.szName, 'TestCPL'）;
      //控制面板中显示的名称
      lStrCpy（PNewCplInfo（lParam2）^.szInfo, 'My Test CPL'）;
      //控制面板中显示的说明信息
```

```
    PNewCplInfo（lParam2）^.szHelpFile[0] := #0;
    Result := 1;
    exit;
  end;
  CPL_SELECT : begin
    Result := 0;
    exit;
  end;
  CPL_DBLCLK : begin
  //双击控制面板中的小应用程序
    WinExec（'Notepad.exe', SW_SHOWNORMAL）;
    //双击后执行记事本
    Result := 1;
    exit;
  end;
  CPL_STOP : begin
    Result := 0;
    exit;
  end;
  CPL_EXIT : begin
    Result := 0;
    exit;
  end else begin
    Result := 0;
    exit;
  end;
 end;
end;

exports CPlApplet name 'CPlApplet';

begin
end.
```

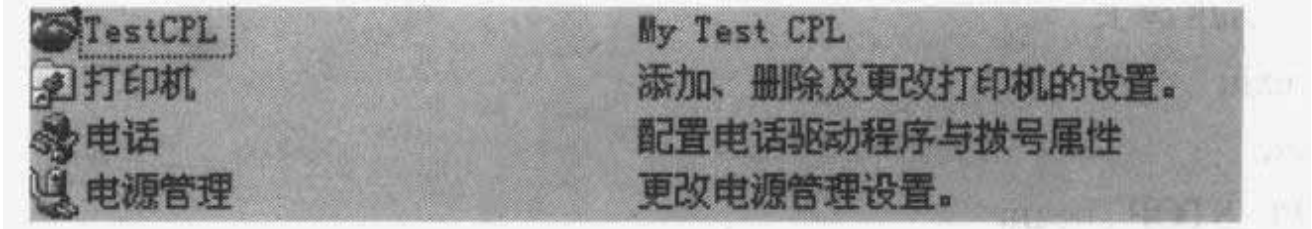
该例子中，关键语句为 exports CPlApplet name 'CPlApplet'，同时对标准的 CPlApplet（）函数进行了定义。该函数中对外界应用程序例如控制面板管理程序所发送的消息进行了处理。为了在控制面板管理程序中显示出您的 applet 的名称、图标以及描述信息则在 CPL_NewInquire 消息中应该将一些数据信息进行定义和填写，例如，定义并填写名称、图标、描述信息以及帮助文档等数据。

编译该动态连接库，将在工程文件所在目录下所获得的动态连接库文件 testcpl.dll 拷贝

至 windows 的系统目录中，并将其后缀名改为.cpl，然后运行控制面板管理程序（如果控制面板管理程序已经打开，则在显示菜单中选择刷新），此时，控制面板的主控制程序将显示该动态连接库的图标、名称等信息，其大图标显示方式以及详细列表显示方式如图 12-1 所示。



（a）大图标方式



（b）详细列表方式

图 12-1　自定义控制面板应用程序



调用控制面板设置功能



遇到难题

前一节讲述了如何定义和创建控制面板应用程序，当定义完成之后，可以通过控制面板管理程序来实现对这些控制面板应用程序的调用和管理。

但有时在自己的应用程序运行时，需要对系统环境提出特殊要求。例如，在 Delphi 数据库应用程序中可能需要进行 BDE（Borland Database Engine）或 ODBC 数据源名称（DSN：Data Source Name）的设置；在网络应用程序中可能需要进行网络配置设置、Modem 属性设置或用户拨号连接的帐号和密码的设置并在系统中保存这些设置；在 CTI（Computer-Telephony Integration ）应用程序中可能需要进行电话拨号属性的设置；在游戏程序中可能需要对多媒体或游戏控制器进行设置；等等。这时候就需要改变 Windows 系统的环境设置，以适应应用程序正常运行的要求。当应用程序不能自行进行环境设置时，可能需要软件使用者来参与和协助完成设置。

如何在自己的应用程序中调用控制面板中的 applet 呢？



钥匙在此

Win 9X/NT 的大多数系统设置是在控制面板中进行的。我们可以要求用户自己打开控制面板并寻找相应的项目进行设置，此时要求用户对控制面板相当熟悉；也可以在程序中自动为用户打开控制面板并选择相应的项目，此时用户所做的只是进行设置。后一种方法无疑更具有友好性，更能提高用户的工作效率。

控制面板的大多数项目设置，都对应于 Windows 的 System 目录下的一个 CPL 文件，如“Internet”属性对应的文件为 InetCpl.cpl，“多媒体”属性对应的文件为 MmSys.cpl 等。通过调用这些文件可以打开相应的属性设置窗口，甚至可以跳到相应的属性页如“Internet”属性的“连接”属性页，这有效地减少了软件用户的操作步骤。

这样，该问题最终归结到如何在应用程序中调用“cpl”的问题。

CPL 文件的调用格式为“RunDLL32.exe Shell32.dll,Control_RunDLL CPLFile.cpl,ItemNo”（可参见注册表中 HKEY_CLASSES_ROOT\cplfile\shell\cplopen \command 的缺省值），其中 CPLFile.cpl 为 CPL 文件， ItemNo 为项目编号。在 Delphi 程序中，可以用 WinExec 或 CreateProcess 函数来实现 CPL 文件调用，从而实现对控制面板的设置项目的调用。执行该语句时，其效果与在控制面板中双击小应用程序效果一样，都会启动该控制面板应用程序。如进行拨号连接的帐号和密码设置要调用“Internet”属性的“连接”属性页，这时可以用语句“WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL InetCpl.cpl,3', SW_SHOWNORM-AL）;”来实现。

又如，解决 2000 年问题中对两位年份输入的解释要调用“区域设置”的“日期”属性页，这时可以用语句： “WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL Intl.cpl,4', SW_SHOWNORMAL）;”来实现。



跟我来

下面将演示如何调用上一节中所定义的控制面板应用程序，即如何在自己的应用程序中调用 TestCPL.cpl。

新建一个应用程序，并在窗体上放置一个按钮，然后在该按钮的单击事件中输入如下代码：

```
procedure TForm1.Button1Click（Sender: TObject）;
begin
  WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL TestCPL.CPL',
    SW_SHOWNORMAL）;
end;
```

单击按钮，将运行自己定义的 TestCPL.CPL，即运行该控制面板应用程序。

对于控制面板中调用其他各项目设置所对应的语句列表如下：

```
//打开控制面板
WinExec（'RunDLL.exe Shell32.DLL,
Control_RunDLL',SW_SHOWNORMAL）;

//32 位 ODBC 数据源管理器
WinExec（'RunDLL32.exe Shell32.dll,
Control_RunDLL ODBCCP32.CPL',SW_SHOWNORMAL）;

//BDE Adminstrator
//没有提供参数区分“Databases”和“Configuration”两个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 BdeAdmin.CPL',SW_SHOWNORMAL）;

//Internet 属性，可有 ItemNo 参数
//“常规”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
InetCpl.cpl,,0', SW_SHOWNORMAL）;
//“安全”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 InetCpl.cpl,,1', SW_SHOWNORMAL）;
//“内容”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 InetCpl.cpl,,2', SW_SHOWNORMAL）;
//“连接”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 InetCpl.cpl,,3', SW_SHOWNORMAL）;
//“程序”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
InetCpl.cpl,,4', SW_SHOWNORMAL）;
//“高级”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
InetCpl.cpl,,5', SW_SHOWNORMAL）;

//电话拨号属性，没有提供参数区分“我的位置”和
“电话语音驱动程序”两个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
Telephon.cpl', SW_SHOWNORMAL）;
```

```
//电源管理设置，没有提供参数区分
“电源方案”和“高级”两个属性页
WinExec（'RunDLL32.exe Shell32.dll,
Control_RunDLL PowerCfg.cpl', SW_SHOWNORMAL）;

//调制解调器设置，没有提供参数区分
“常规”和“诊断”两个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control
_RunDLL Modem.cpl', SW_SHOWNORMAL）;

//多媒体设置，可有 ItemNo 参数
//“音频”属性页
WinExec（'RunDLL32.exe Shell32.dll,
Control_RunDLL Mmsys.cpl,,0', SW_SHOWNORMAL）;
//“视频”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_
RunDLL Mmsys.cpl,,1', SW_SHOWNORMAL）;
//“MIDI”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_
RunDLL Mmsys.cpl,,2', SW_SHOWNORMAL）;
//“CD 音乐”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_
RunDLL Mmsys.cpl,,3', SW_SHOWNORMAL）;
//“设备”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Mmsys.cpl,,4', SW_SHOWNORMAL）;

//辅助选项设置，可有 ItemNo 参数
//“键盘”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 access.cpl,,1',SW_SHOWNORMAL）;
//“声音”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 access.cpl,,2', SW_SHOWNORMAL）;
//“显示”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 access.cpl,,3', SW_SHOWNORMAL）;
//“鼠标”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 access.cpl,,4', SW_SHOWNORMAL）;
```

```
//“常规”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 access.cpl,,5', SW_SHOWNORMAL）;

//密码设置，没有提供参数区分“更改密码”和
“修改用户配置”两个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Password.cpl', SW_SHOWNORMAL）;

//区域设置，可有 ItemNo 参数
//“区域设置”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Intl.cpl,,0', SW_SHOWNORMAL）;
//“数字”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Intl.cpl,,1', SW_SHOWNORMAL）;
//“货币”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Intl.cpl,,2', SW_SHOWNORMAL）;
//“时间”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Intl.cpl,,3', SW_SHOWNORMAL）;
//“日期”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 Intl.cpl,,4', SW_SHOWNORMAL）;

//日期/时间设置，可有 ItemNo 参数
//“日期和时间”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 TimeDate.cpl,,0', SW_SHOWNORMAL）;
//“时区”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
TimeDate.cpl,,1', SW_SHOWNORMAL）;

//鼠标设置，没有提供参数区别“按钮”、“指针”和“移动”三个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
Main.cpl', SW_SHOWNORMAL）;

//添加/删除程序设置，可有 ItemNo 参数
//“安装/卸载”属性页
```

```
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 AppWiz.cpl,,1', SW_SHOWNORMAL）;
//“Windows 安装程序”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 AppWiz.cpl,,2', SW_SHOWNORMAL）;
//“启动盘”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
AppWiz.cpl,,3', SW_SHOWNORMAL）;

//网络设置，没有提供参数区别“配置”、“标识”和“访问控制”三个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 NetCpl.cpl', SW_SHOWNORMAL）;

//系统设置，可有 ItemNo 参数
//“常规”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 SysDm.cpl,,0', SW_SHOWNORMAL）;
//“设备管理器”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 SysDm.cpl,,1', SW_SHOWNORMAL）;
//“硬件配置文件”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
SysDm.cpl,,2', SW_SHOWNORMAL）;
//“性能”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
SysDm.cpl,,3', SW_SHOWNORMAL）;

//显示设置， 可有 ItemNo 参数
//“背景”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
desk.cpl,,0', SW_SHOWNORMAL）;
//“屏幕保护程序”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
desk.cpl,,1', SW_SHOWNORMAL）;
//“外观”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 desk.cpl,,2', SW_SHOWNORMAL）;
//“设置”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
desk.cpl,,3', SW_SHOWNORMAL）;
```

```
//游戏控制器设置，可有 ItemNo 参数
//“一般”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
Joy.cpl,,0', SW_SHOWNORMAL）;
//“高级”属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
Joy.cpl,,1', SW_SHOWNORMAL）;

//扫描仪与数字相机设置，没有提供参数区分“设备”和“记录设置”两个属性页
WinExec（'RunDLL32.exe Shell32.dll,Control_RunDLL
 StiCpl.cpl', SW_SHOWNORMAL）;
```



检测即插即用硬件发生的变化



遇到难题

例如，当您的资源管理器运行时在光驱中插入光盘，则此时资源管理器就会自动获得光盘的相应信息，例如卷标，并将之进行刷新。如果此时再将光盘取走，则资源管理器又会自动刷新，光驱驱动器中又显示出无光盘的状态。又如，超级解霸的光驱检测器在有光盘插入时会自动得到通知，如果是 CD、VCD 或者 DVD 的话则会自动启动自己的播放器。那么，如何将这一功能添加到应用程序中，即如果当即插即用的硬件设备发生变化时，您的程序如何感知呢？



钥匙在此

当计算机即插即用设备配置或者状态发生更改时，系统将使用 WM_DEVICECHANGE 设备消息通知应用程序或者设备的驱动程序。该消息的参数如下所示：

```
Event = （UINT） wParam;
dwData = （DWORD） lParam;
```

其中参数 Event 为事件类型，该参数可以是以下值中的一个，见表 12-1。

表 12-1　　WM_DEVICECHANGE 消息事件类型

值	描　述
DBT_CONFIGCHANGECANCELED	改变当前配置的请求已经被取消了
DBT_CONFIGCHANGED	当前的配置已经被改变
DBT_DEVICEARRIVAL	设备已经被插入并且现在可以使用了
DBT_DEVICEQUERYREMOVE	请求允许移动设备。任何一个应用程序都可以拒绝这个请求，并且取消移动
DBT_DEVICEQUERYREMOVEFAILED	移动设备的请求已经被取消了
DBT_DEVICEREMOVECOMPLETE	设备已经移走
DBT_DEVICEREMOVEPENDING	设备将要移开。不能被拒绝
DBT_DEVICETYPESPECIFIC	一个特殊设备事件已经发生
DBT_QUERYCHANGECONFIG	请求允许改变当前的配置
DBT_USERDEFINED	用户定义的消息

参数 dwData 为载有指定事件数据的结构的地址。它的含义取决于所给定的事件。例如，对于 DBT_DEVICEARRIVAL 事件而言，该参数为 DEV_BROADCAST_HDR 结构类型变量的地址。该变量标识了所插入的设备。

该消息的返回值，在同意请求的操作时为 True，在拒绝指定的操作时则为 QUERY_BROADCAST_DENY。

对于那些提供软件可控特点的设备来讲，例如弹出或者锁定，操作系统典型地会发送一个 DBT_DEVICEREMOVEPENDING 消息通知应用程序和设备驱动程序来结束对这些设备的使用。如果操作系统强烈要求移开这一设备，那么在这样做之前不会发送 DBT_DEVICEQUERYREMOVE 消息。



跟我来

下列代码演示了上述事件的应用。新建一个应用程序，并在该程序中创建一个 WM_DEVICECHANGE 消息处理过程，整个代码参见清单 12-2。

清单 12-2　检测硬件配置或状态发生的变化

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class （TForm）
  private
    { Private declarations }
```

```
    procedure WMDeviceChange（var Message: TMessage）;
      message WM_DEVICECHANGE;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

const DBT_DEVICEARRIVAL = $8000;
const DBT_DEVICEQUERYREMOVE = $8001;
const DBT_DEVICEQUERYREMOVEFAILED = $8002;
const DBT_DEVICEREMOVEPENDING = $8003;
const DBT_DEVICEREMOVECOMPLETE = $8004;
const DBT_DEVICETYPESPECIFIC = $8005;
const DBT_CONFIGCHANGED = $0018;

procedure TForm1.WMDeviceChange（var Message: TMessage）;
var
  s : string;
begin
{Do Something here}
  case Message.wParam of
    //一个设备已被插入并且现在可用
    DBT_DEVICEARRIVAL :
      s := 'A device has been inserted and is now available';
    //请求允许移动设备
    DBT_DEVICEQUERYREMOVE: begin
      s := 'Permission to remove a device is requested';
      ShowMessage（s）;
     {True grants premission}
      Message.Result := integer（true）;
      exit;
    end;
    //移动设备的请求已经被取消了
    DBT_DEVICEQUERYREMOVEFAILED :
```

```
    s := 'Request to remove a device has been canceled';
//设备将要移开，不能被拒绝
DBT_DEVICEREMOVEPENDING :
    s := 'Device is about to be removed';
//设备已经移走
DBT_DEVICEREMOVECOMPLETE :
    s := 'Device has been removed';
//一个特殊设备事件已经发生
DBT_DEVICETYPESPECIFIC :
    s := 'Device-specific event';
//当前的配置已经被改变
DBT_CONFIGCHANGED :
        s:= 'Current configuration has changed'
      else s := 'Unknown Device Message';
    end;
    ShowMessage（s）;
    inherited;
end;

end.
```

运行该应用程序，当在光驱中插入一张光盘时将激发“设备被插入”事件，此时，该应用程序将会弹出“A device has been inserted and is now available”对话框；当将光驱中的盘取走时，则会弹出“Device has been removed”对话框。



直接控制 I/O



遇到难题

Delphi 以其强大的功能及方便快捷的程序设计而为广大程序员所喜爱。但是在用它来编写工业控制程序时，需要对计算机所连接的外部设备进行操作，即直接对 I/O 地址进行读写操作，这时它就显得有些美中不足了。但在 Delphi 中提供了解决此问题的方法。



钥匙在此

针对这一问题，可以使用内嵌汇编的方式编写应用程序。内嵌汇编可以通过 asm 语句实现，该语句格式如下：

asm statementList end

这里，statementList 为一系列通过分号隔开以及以行结束符结尾的或者是 Pascal 注释所组成的汇编语句。在 asm 语句中的注释必须符合 Pascal 对象类型。至于内嵌汇编请参见有关章节。



跟我来

下面定义了一个模块 PORT95. PAS，可方便地实现直接对 I/O 地址的读写操作，代码简捷且执行速度较快。

使用时只要将 PORT95.PAS 加到工程文件中，并在 users 中加上 Port95，就可以在应用程序中直接对 I/O 端口进行操作。

具体的实现方法及 PORT95.PAS 的源代码见清单 12-3。

清单 12-3　I/O 读写操作单元

```
unit Port95;
interface
function PortReadByte（Addr:Word） : Byte;
function PortReadWord（Addr:Word） : Word;
function PortReadWordLS（Addr:Word） : Word;
procedure PortWriteByte（Addr:Word; Value:Byte）;
procedure PortWriteWord（Addr:Word; Value:Word）;
procedure PortWriteWordLS（Addr:Word; Value:Word）;
implementation
{*
* Port Read byte function
*Parameter:port address
*Return: byte value from given port
*}
function PortReadByte（Addr:Word） : Byte; assembler; register;
asm
MOV DX,AX
IN AL,DX
end;
{*
```

```
* HIGH SPEED Port Read Word function
* Parameter: port address
* Return: word value from given port
* Comment:may problem with some cards and computers that
can't to access whole word, usualy it works.
*}
function PortReadWord（Addr:Word） : Word; assembler; register;
asm
MOV DX,AX
IN AX,DX
end;
{*
* LOW SPEED Port Read Word function
* Parameter: port address
*Return:word value from given port
*Comment:work in cases,only to adjust DELAY if need
*}
function PortReadWordLS（Addr:Word） : Word; assembler; register;
const
Delay = 150;
// depending of CPU speed and cards speed
asm
MOV DX,AX
IN AL,DX
//read LSB port
MOV ECX,Delay
@1:
LOOP @1 //delay between two reads
XCHG AH,AL
INC DX
//port+1
IN AL,DX //read MSB port
XCHG AH,AL //restore bytes order
end;
{* Port Write byte function*}
procedure PortWriteByte（Addr:Word; Value:Byte）; assembler; register;
asm
XCHG AX,DX
OUT DX,AL
end;
{*
```

```
* HIGH SPEED Port Write word procedure
* Comment:may problem with some cards and computers that
can't to access whole word, usualy it works.
*}
procedure PortWriteWord（Addr:word; Value:word）; assembler; register;
asm
XCHG AX,DX
OUT DX,AX
end;
{*
* LOW SPEED Port Write Word procedure
*}
procedure PortWriteWordLS（Addr:word; Value:word）; assemb
ler; register;
const
Delay = 150;
// depending of CPU speed and cards speed
asm
XCHG AX,DX
OUT DX,AL
MOV ECX,Delay
@1:
LOOP@1
XCHG AH,AL
INC DX
OUT DX,AL
end;
end. //单元结束
```



获取系统一般信息



遇到难题

Delphi 以其优良的可视化编程、灵活的 Windows API 接口以及丰富的底层操作而越来

越受到编程爱好者的青睐。在 Delphi 中，通过调用 Windows API 可以很方便地获取系统信息，这有助于我们编写出更好的 Windows 应用程序。

在控制面板中双击“系统”图标可以得到许多系统信息，如内存、CPU 以及 OS 版本等信息。如何在您的应用程序中得到这些信息呢？



钥匙在此

系统信息一般包括了内存、CPU 以及 OS 等信息。下面分别进行阐述。

1. 用 GlobalMemoryStatus（）函数获取内存使用信息

GlobalMemoryStatus（）用于获取关于当前可用内存的信息。该函数将返回关于物理以及虚拟内存的信息，在 windows.pas 中是这样进行声明的：

```
procedure GlobalMemoryStatus（var lpBuffer: TMemoryStatus）; stdcall;
```

其中参数 lpBuffer 为内存状态 TMemoryStatus 结构变量的指针。在调用该函数之前，调用的处理过程应该设置 TmemoryStatus 的 dwLength 成员。关于 TmemoryStatus 结构的声明及意义如下：

```
TMEMORYSTATUS = record
  dwLength: DWORD;//结构变量的大小 sizeof（TMEMORYSTATUS）
  dwMemoryLoad: DWORD;//使用中的内存百分比
  dwTotalPhys: DWORD;//物理内存字节数
  dwAvailPhys: DWORD;//可用的物理内存字节数
  dwTotalPageFile: DWORD;//页文件大小的字节数
  dwAvailPageFile: DWORD;//可用的页文件字节数
  dwTotalVirtual: DWORD;//地址空间的用户字节数，即虚拟空间字节数
  dwAvailVirtual: DWORD;//可用的用户字节数
end;
```

其中参数 dwLength 表示该结构变量的大小，在调用相应的函数之前应当设置其值；参数 dwMemoryLoad 指定了当前内存使用的百分比，如果为 0，表示无内存被使用，100 表示所有的内存被使用；参数 dwTotalPhys 为物理内存字节数；参数 dwAvailPhys 为当前可用的物理内存字节数；参数 dwTotalPageFile 表示能存储在页文件中数据的字节数；参数 dwAvailPageFile 表示可用的页文件字节数；参数 dwTotalVirtual 表示地址空间的用户字节数，即虚拟空间字节数；参数 dwAvailVirtual 表示可用的用户字节数。

下面一个例子演示了该函数是如何使用的。

```
var
    MemStatus: TMEMORYSTATUS; //定义内存结构变量
Begin
    MemStatus.dwLength := sizeof（TMEMORYSTATUS）;
    GlobalMemoryStatus（MemStatus）; //返回内存使用信息
    showmessage（format（'可用内存: %d KB:  共有内存:%dKB',[MemStatus.dwAvailPhys
```

```
        div 1024,MemStatus.dwTotalPhys div 1024]）;//将内存信息显示出来
    end;
```

2. 用 GetSystemInfo（）函数获取 CPU 信息

GetSystemInfo（）函数返回关于当前系统的信息。该函数在 windows.pas 中的声明格式如下：

```
procedure GetSystemInfo（var lpSystemInfo: TSystemInfo）; stdcall;
```

其中，参数 lpSystemInfo 为 TsystemInfo 结构变量指针，用于被填充信息数据。TsystemInfo 结构变量载有关于当前计算机的信息，这包括了系统结构、处理器类型以及该系统中处理器的数量、页大小和其他类似的信息。该类型变量中与 CPU 类型有关的域成员主要有 wProcessorArchitecture、dwProcessorType、dwNumberOfProcessors、wProcessorLevel 和 wProcessorRevision 等，现对这些域字段解释如表 12-2 所示。

表 12-2　　TsystemInfo 部分域

域　名	描　述
wProcessorArchitecture	指定系统处理器的结构。这个值可以取以下值之一： PROCESSOR_ARCHITECTURE_INTEL Windows NT only: PROCESSOR_ARCHITECTURE_MIPS Windows NT only: PROCESSOR_ARCHITECTURE_ALPHA Windows NT only: PROCESSOR_ARCHITECTURE_PPC Windows NT only: PROCESSOR_ARCHITECTURE_UNKNOWN
dwProcessorType	Windows 95: 指定系统中处理器类型 Windows NT: 该值不再与之相关，只是用于与 Windows 95 及以前的 NT 版本兼容。 该成员可以取以下值之一： PROCESSOR_INTEL_386 PROCESSOR_INTEL_486 PROCESSOR_INTEL_PENTIUM Windows NT only: PROCESSOR_MIPS_R4000 Windows NT only: PROCESSOR_ALPHA_21064
wProcessorLevel	Windows 95:该成员不被使用 Windows NT:指定系统所依赖的结构处理器级别 如果 wProcessorArchitecture 是 PROCESSOR_ARCHITECTURE_INTEL 时，wProcessorLevel 可以取以下值之一： 值　含义 3　Intel 80386 4　Intel 80486 5　Pentium 如果 wProcessorArchitecture 是 PROCESSOR_ARCHITECTURE_MIPS 时，wProcessorLevel 是 00xx 的形式，这里 xx 为 8 位执行数（bits 8-15 of the PRId register）。该成员可以取以下值之一： 值　含义 0004　MIPS R4000 如果 wProcessorArchitecture 为 PROCESSOR_ARCHITECTURE_ALPHA 时，ProcessorLevel 则是 xxxx 的形式，这里 xxxx 为 16 位处理器版本数（the low-order 16 bits of a version number from the firmware）。该成员可以取以下

续表

域　名	描　述
	值之一： 十进制值　含义 21064　Alpha 21064 21066　Alpha 21066 21164　Alpha 21164 如果 wProcessorArchitecture 是 PROCESSOR_ARCHITECTURE_PPC 时，wProcessorLevel 是 xxxx 形式，这里 xxxx 为 16 位处理器版本数（the high-order 16 bits of the Processor Version Register）。该成员可以取以下值之一： 十进制值　含义 1　PPC 601 3　PPC 603 4　PPC 604 6　PPC 603+ 9　PPC 604+ 20 PPC 620
wProcessorRevision	Windows 95:该成员没有使用 Windows NT:指定一个独立结构处理器的修订值。具体内容参见 Win32 API 中的 SYSTEM_INFO

下列例子描述了该结构变量以及函数的使用方法。

```
var
    SysInfo: TSYSTEMINFO;
begin
    case SysInfo.dwProcessorType of
    386:showmessage（format（'%d%s',[SysInfo.dwNumberOfProcessors,
    'Intel80386']））；
    486:showmessage（format（'%d%s',[SysInfo.dwNumberOfProcessors,
    'Intel 80486']））；
    586:showmessage（format（'%d%s',[SysInfo.dwNumberOfProcessors,
    'Intel Pentium']））；
      end;
  end;
```

3. 用 GetVersionEx（）获取操作系统信息

要获取当前运行的 Windows 版本，可以调用 GetVersionEx（）。该函数只有一个 TOSVersionInfo 类型的变量参数。TOSVersionInfo 是这样声明的：

```
  TOSVersionInfo = record
    dwOSVersionInfoSize: DWORD;
    dwMajorVersion: DWORD;
    dwMinorVersion: DWORD;
    dwBuildNumber: DWORD;
    dwPlatformId: DWORD;
```

```
    szCSDVersion: array[0..127] of AnsiChar; { Maintenance string for PSS usage }
  end;
```

其中，域 dwOSVersionInfoSize 为该结构变量所占据的类型大小，在调用该函数之前应当用 sizeof（TOSVersionInfo）设置其大小。

域 dwMajorVersion 为操作系统的主版本号。假设 OS 版本为 4.0，则这个域的值为 4。

域 dwMinorVersion 为操作系统的副版本号。假设 OS 版本为 4.0，则这个域的值为 0。

域 dwBuildNumber 为操作系统的建立序号。

域 dwPlatformId 表明当前的 Win32 平台的类型，可以取以下值：

值	描述
VER_PLATFORM_WIN32s	Win32s on Windows 3.1.
VER_PLATFORM_WIN32_WINDOWS	Win32 on Windows 95.
VER_PLATFORM_WIN32_NT	Win32 on Windows NT.

域 szCSDVersion 可能包含了操作系统的其他信息，不过它往往是空的。

下列代码演示了如何使用 TOSVersionInfo 结构变量和 GetVersionEx（）函数。

```
var
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf（VI）;
  GetVersionEx（VI）;
  Showmessage（Format（'OS 版本%d.%d', [dwMajorVersion, dwMinorVersion]））;
End;
```



跟我来

下面看一个完整的例子。先创建一个应用程序，并在其上放置三个按钮以及一个 TlistBox 控件，其代码参见清单 12-4。

清单 12-4　获取系统一般信息

```
unit infou;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ComCtrls, Buttons;

type
```

```
TInfoVariety =  （ivMemory, ivSystem, ivEnvironment）;

TinfoForm = class（TForm）
   infoLB: TListBox;
   BitBtn1: TBitBtn;
   BitBtn2: TBitBtn;
   BitBtn3: TBitBtn;
   procedure GetOSVerInfo;
   procedure GetSysInfo;
   procedure GetDirInfo;
   procedure BitBtn1Click（Sender: TObject）;
   procedure BitBtn2Click（Sender: TObject）;
   procedure BitBtn3Click（Sender: TObject）;
  public
     procedure ShowMemStatus;
     procedure ShowSysInfo;
     procedure ShowEnvironment;
     procedure ShowInformation（Variety: TInfoVariety）;
end;
var
  infoForm:TInfoForm;


implementation

{$R *.DFM}

procedure TInfoForm.ShowInformation（Variety: TInfoVariety）;
begin
       case Variety of
          ivMemory: ShowMemStatus;
          ivSystem: ShowSysInfo;
          ivEnvironment: ShowEnvironment;
       end;
end;

const
  { Memory status strings }
  SMemUse    ='正使用内存：%d%%';
  STotMem    ='总共内存：$%.8x bytes';
```

```
SFreeMem    = '可用的物理内存：$%.8x bytes';
STotPage    = '总共的页文件大小：$%.8x bytes';
SFreePage = '可用的页文件大小：$%.8x bytes';
STotVirt    = '总共的虚拟内存：$%.8x bytes';
SFreeVirt = '可用的虚拟内存：$%.8x bytes';

{ OS version info strings }
SOSVer       = 'OS 版本%d.%d';
SBuildNo    = '版本序号：%d';
SOSPlat      = '平台：%s';
SOSWin32s = 'Windows 3.1x running Win32s';
SOSWin95    = 'Windows 95';
SOSWinNT    = 'Windows NT';

{ System info strings }
SProc        = 'Processor Arhitecture：%s';
SPIntel      = 'Intel';
SPageSize = 'Page Size：$%.8x bytes';
SMinAddr    = 'Minimum Application Address：$%p';
SMaxAddr    = 'Maximum Application Address：$%p';
SNumProcs = 'Number of Processors：%d';
SAllocGra = 'Allocation Granularity：$%.8x bytes';
SProcLevl = 'Processor Level：%s';
SIntel3      = '80386';
SIntel4      = '80486';
SIntel5      = 'Pentium';
SIntel6      = 'Pentium Pro';
SProcRev    = 'Processor Revision：%.4x';

  { Directory strings }
  SWinDir      = 'Windows directory：%s';
  SSysDir      = 'Windows system directory：%s';
  SCurDir      = 'Current directory：%s';

//显示内存信息
procedure TInfoForm.ShowMemStatus;
var
  MS: TMemoryStatus;
begin
  MS.dwLength := SizeOf（MS）;
```

```
GlobalMemoryStatus（MS）;
with InfoLB.Items, MS do
begin
  Clear;
  Add（Format（SMemUse, [dwMemoryLoad]））;
  Add（Format（STotMem, [dwTotalPhys]））;
  Add（Format（SFreeMem, [dwAvailPhys]））;
  Add（Format（STotPage, [dwTotalPageFile]））;
  Add（Format（SFreePage, [dwAvailPageFile]））;
  Add（Format（STotVirt, [dwTotalVirtual]））;
  Add（Format（SFreeVirt, [dwAvailVirtual]））;
end;
  caption:='内存信息';
end;

//获取操作系统的信息
procedure TInfoForm.GetOSVerInfo;
var
  VI: TOSVersionInfo;
begin
  VI.dwOSVersionInfoSize := SizeOf（VI）;
  GetVersionEx（VI）;
  with InfoLB.Items, VI do
  begin
    Clear;
    Add（Format（SOSVer, [dwMajorVersion, dwMinorVersion]））;
    Add（Format（SBuildNo, [LoWord（dwBuildNumber）]））;
    case dwPlatformID of
      VER_PLATFORM_WIN32S: Add（Format（SOSPlat, [SOSWin32s]））;
      VER_PLATFORM_WIN32_WINDOWS: Add（Format（SOSPlat, [SOSWin95]））;
      VER_PLATFORM_WIN32_NT: Add（Format（SOSPlat, [SOSWinNT]））;
    end;
  end;
end;

//获取系统信息
procedure TInfoForm.GetSysInfo;
var
  SI: TSystemInfo;
begin
```

```
GetSystemInfo（SI）;
with InfoLB.Items, SI do
begin
  Add（Format（SProc, [SPIntel]））;
  Add（Format（SPageSize, [dwPageSize]））;
  Add（Format（SMinAddr, [lpMinimumApplicationAddress]））;
  Add（Format（SMaxAddr, [lpMaximumApplicationAddress]））;
  Add（Format（SNumProcs, [dwNumberOfProcessors]））;
  Add（Format（SAllocGra, [dwAllocationGranularity]））;
  case wProcessorLevel of
    3: Add（Format（SProcLevl, [SIntel3]））;
    4: Add（Format（SProcLevl, [SIntel4]））;
    5: Add（Format（SProcLevl, [SIntel5]））;
    6: Add（Format（SProcLevl, [SIntel6]））;
    else Add（Format（SProcLevl, [IntToStr（wProcessorLevel）]））;
    end;
  end;
end;

procedure TInfoForm.GetDirInfo;
var
  S: array[0..MAX_PATH] of char;
begin
  { 获取 Windows 目录 }
  GetWindowsDirectory（S, SizeOf（S））;
  InfoLB.Items.Add（Format（SWinDir, [S]））;
  { 获取 Windows 系统目录}
  GetSystemDirectory（S, SizeOf（S））;
  InfoLB.Items.Add（Format（SSysDir, [S]））;
  {获得应用程序的当前路径}
  GetCurrentDirectory（SizeOf（S）, S）;
  InfoLB.Items.Add（Format（SCurDir, [S]））;
end;

procedure TInfoForm.ShowSysInfo;
begin
  GetOSVerInfo;
  GetSysInfo;
  GetDirInfo;
  Caption:= '系统信息';
```

```
end;

procedure TInfoForm.ShowEnvironment;
var
  EnvPtr, SavePtr: PChar;
begin
  EnvPtr := GetEnvironmentStrings;
  SavePtr := EnvPtr;
  InfoLB.Items.Clear;
  repeat
    InfoLB.Items.Add（StrPas（EnvPtr））;
    inc（EnvPtr, StrLen（EnvPtr） + 1）;
  until EnvPtr^ = #0;
  FreeEnvironmentStrings（SavePtr）;
  Caption:= '当前的环境变量';
end;

procedure TinfoForm.BitBtn1Click（Sender: TObject）;
begin
    ShowInformation（ivMemory）;
end;

procedure TinfoForm.BitBtn2Click（Sender: TObject）;
begin
    ShowInformation（ivsystem）;
end;

procedure TinfoForm.BitBtn3Click（Sender: TObject）;
begin
    ShowInformation（ivEnvironment）;
end;

end.
```

其中，ShowInformation（）为总控制函数，根据其参数的不同选择可以显示不同的信息。当单击按钮 1 时，通过总控制程序调用 ShowMemStatus（）函数来获取内存信息，结果如图 12-2 所示。

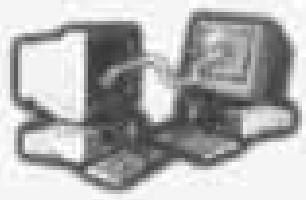
当单击按钮 2 时，通过总控程序调用 GetSysInfo（）函数来获取系统信息，如 CPU 的信息等。结果如图 12-3 所示。



图 12-2 获取内存信息



图 12-3 获取系统信息



检查环境变量



遇到难题

为了在多个应用程序间或应用程序内部共享一些数值或者设置，除了使用注册表之外，

还有一个很好的方式，那就是使用环境变量。例如，当安装程序进行安装时，由用户录入一些交互信息存入自动批处理文件中，譬如，SETS、PATH 和 PROMPT 等信息。然后通过重新启动操作系统来获取这部分信息。那么如何在自己的应用程序中访问这些变量呢？



钥匙在此

要获取环境变量的情况，诸如 SETS、PATH 和 PROMPT 等是很简单的，这可以调用 GetEnvironmentStrings（）函数来实现。这个函数没有参数，它返回一组字符串，彼此以 NULL 字符隔开，即一个字符串后跟一个 NULL，然后又是一个字符串，再跟一个 NULL，一直到遇到连续两个 NULL（#0#0）为止。下面的代码显示了怎样调用 GetEnvironmentStrings（）函数。

```
var
   EnvPtr, SavePtr: PChar;
   S:string;
begin
   EnvPtr := GetEnvironmentStrings;
   SavePtr := EnvPtr;
   repeat
      s:=s+StrPas（EnvPtr）;
      inc（EnvPtr, StrLen（EnvPtr） + 1）;
   until EnvPtr^ = #0;
   FreeEnvironmentStrings（SavePtr）;
   Showmessage（s）;
   End;
```

该例子中，由于在返回的字符串中一个字符接着一个 Null 符号，因此，为了将其中的所有字符信息取出来，在移动字符指针时应移动两个字节，当获得最终的 Null 时，停止提取字符。注意，最后不要忘了调用 FreeEnvironmentStrings（）函数来释放环境变量字符串。



跟我来

具体例子参见前一节清单 12-4。当单击按钮 3 时，获得结果如图 12-4 所示。



图 12-4　获取所有环境变量



再送礼包

GetEnvironmentStrings（）无法获取或者设置单个环境变量的值。如果要获取或者设置单个环境变量的值，请调用 GetEnvironmentVariable（）或者 SetEnvironmentVariable（）函数。



设置桌面墙纸



遇到难题

在 IE 或者其他 Web 浏览器中，当您看到一幅漂亮的图片时，可能希望将其设为桌面的背景。选择一幅漂亮的图片，然后在该图片上点击鼠标右键，选择相应选项就可以将选中的图片作为背景图片。有时，在安装了许多商业软件后，也可以选择产品的一些漂亮图片作为背景墙纸来显示。同样，在您的应用程序中也可以将选定的图片作为背景图案。



钥匙在此

调用 Windows API 函数 SystemParametersInfo（），并将 SPI_SETDESKWALLPAPER 参

数随着指定的位图一起传给该函数，即可实现将选择的位图作为背景的功能。该函数在 Windows.pas 中声明格式为：

```
function SystemParametersInfo（uiAction, uiParam: UINT;
  pvParam: Pointer; fWinIni: UINT）: BOOL; stdcall;
```

该函数可以查询和设置系统参数。当进行设置时，可以更改系统信息。其中参数 uiAction 指定了要查询或者要设置的参数，其参数值可以参见表 12-3。参数 uiParam 以及 pvParam 的含义取决于要查询或者设置的系统参数。参数 fWinIni 在进行系统参数的设置时，应当用来指定用户信息是否被修改，或者是否将设置更改的消息 WM_SETTINGCHANGE 广播给所有启动的窗体程序，该参数可以是 0，也可以是表 12-4 中值之一。

表 12-3　　系统参数 uiAction 取值

值	含义
SPI_SETDESKWALLPAPER	设置桌面墙纸。pvParam 参数必须是载有位图的文件名
SPI_SETDOUBLECLICKTIME	设置双击鼠标的时间。uiParam 参数代表时间值
其余参见 SystemParametersInfo（）函数	

表 12-4　　参数 fWinIni 取值

值	动作描述
SPIF_UPDATEINIFILE	将新的系统参数设置写入用户的系统信息中
SPIF_SENDCHANGE	在修改了系统信息后广播 WM_SETTINGCHANGE 消息
SPIF_SENDWININICHANGE	与 SPIF_SENDCHANGE 一样

如果该函数执行成功，则返回值为非 0，若失败则返回 0。该函数主要用于设置定制系统环境，例如可以用于控制面板应用程序。

下面看一个调用该函数的例子。

```
SystemParametersInfo（SPI_SETDESKWALLPAPER,
                      0,
                      PChar（'C:\SOMEPATH\SOME.BMP'）,
                      SPIF_SENDWININICHANGE）;
```



跟我来

新建一个应用程序，然后将一按钮放置在窗体上，并输入以下代码，见清单 12-5。

清单 12-5　设置墙纸

```
procedure TForm1.Button1Click（Sender: TObject）;
var
  s: string;
begin
   if OpenPictureDialog1.Execute then
       s := OpenPictureDialog1.FileName;
```

```
        SystemParametersInfo（SPI_SETDESKWALLPAPER, 0, PChar（s）, 0）;
        //将指定的位图作为墙纸显示
    end;
end;
```



捕获系统按键



遇到难题

记得以前到图书馆进行文献资料的查询时，由于检索是要收费的，因此会根据检索了多少条目来记费。当时作为学生也够穷的，想了一个坏招，直接按 PrintScreen 进行拷屏输入到剪贴板中，然后由附件中的画笔将剪贴板中的内容取出，存成文件然后带走。但如果是一个好的检索软件的话，则完全应该杜绝这种事情的发生，例如，在应用程序中可以截获一些系统按键，并对其进行处理。那么在应用程序中如何实现这一功能呢？



钥匙在此

为了截取系统按键需要做两件事，首先需要调用 API 函数 RegisterHotKey（）为当前线程定义一个热键，然后定义一个 WM_HOTKEY 热键消息处理函数来截获用户所按的热键。下面进行详细阐述。

1. 定义一个热键

调用 RegisterHotKey（）函数可以注册一个热键。当该键按下之后，系统将从所有线程中寻找一个匹配。在找到匹配之后，系统将发送 WM_HOTKEY 消息至注册热键线程的消息队列中。这个消息被放置在队列之首，所以会通过下一个消息循环而将其移去。该函数在 windows.pas 单元中的声明格式如下：

function RegisterHotKey（hWnd: HWND; id: Integer; fsModifiers, vk: UINT）: BOOL; stdcall;

参数 hWnd 为获取热键按下通知的窗口句柄，如果该参数是空，WM_HOTKEY 消息将被发送到调用线程的消息队列中，并且必须在消息循环中得到处理。

参数 id 为热键的 ID 号。在调用线程中不应该有其他热键具有同样的 ID 号，一个应用程序指定的 ID 号值应当在 0x0000 到 0xBFFF 之间。为了避免冲突，可以通过 GlobalAddAtom（）函数来取得热键的 ID 号。

参数 fsModifiers 指定了与 nVirtKey 参数进行组合时必须按下的键，该参数的值可以为

下列值的组合。

值	意 义
MOD_ALT	任意一个【ALT】必须被按下
MOD_CONTROL	任意一个【CTRL】键必须按下
MOD_SHIFT	任意一个【Shift】键必须按下

参数 vk 指定热键的虚拟键值。

返回值：如果该函数运行成功，则返回为非 0，否则为 0。当程序退出时应当将注册的热键通过调用 UnregisterHotKey（）函数解除。

例如，将 PrintScreen 键注册为热键，代码如下：

```
Const
    id_SnapShot = 101;
begin
  //将 PrintScreen 键注册为一个热键，并与 ALT 键进行组合
  RegisterHotKey（Form1.Handle,
                  id_SnapShot,
                  MOD_ALT,
                  VK_SNAPSHOT）;
end;
```

其中，VK_SNAPSHOT 为 PrintScreen 键的虚拟键值。

2. 定义热键消息处理函数

WM_HOTKEY 消息在用户按了通过 RegisterHotKey 函数所注册的热键时将被发送。该消息被放置在注册该热键的线程消息队列的首位。该消息类型在 messages.pas 中声明如下：

```
TWMHotKey = packed record
    Msg: Cardinal;//消息常量
    HotKey: Longint;//热键的 ID 号。
    Unused: Longint;//
    Result: Longint;
end;
```

参数 idHotKey 为 wParam 的值，它指定了产生消息的热键的 ID 号。如果该消息是被系统定义的热键所产生，则该参数将是下列值之一。

值	含义
IDHOT_SNAPDESKTOP	"snap desktop"热键被按下
IDHOT_SNAPWINDOW	"snap window"热键被按下

例如，现在定义一个 WM_HOTKEY 消息处理函数。

```
procedure WMHotKey（var Msg : TWMHotKey）; message WM_HOTKEY;
```

```
procedure TForm1.WMHotKey （var Msg : TWMHotKey）;
const
id_SnapShot = 101;
begin
   if Msg.HotKey = id_SnapShot then
      ShowMessage（'GotIt'）;
end;
```



跟我来

下面是一个截获 ALT+PrintScreen 组合键的完整例子，代码请参见清单 12-6。

清单 12-6　截获 Alt+PrintScreen 组合键

```
unit Unit1;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
TForm1 = class（TForm）
      procedure FormCreate（Sender: TObject）;
      procedure FormDestroy（Sender: TObject）;
   private
      { Private declarations }
      //声明一个消息处理函数
      procedure WMHotKey（var Msg : TWMHotKey）; message WM_HOTKEY;
   public
      { Public declarations }
   end;

var
   Form1: TForm1;

implementation
```

```
{$R *.DFM}
const
    id_SnapShot = 101;

//WM_HOTKEY 消息的处理函数
procedure TForm1.WMHotKey （var Msg : TWMHotKey）;
begin
   if Msg.HotKey = id_SnapShot then
      ShowMessage（'GotIt'）;
end;

procedure TForm1.FormCreate（Sender: TObject）;
begin
   //将 PrintScreen 键注册为一个热键，并与 ALT 键进行组合
   RegisterHotKey（Form1.Handle,
                   id_SnapShot,
                   MOD_ALT,
                   VK_SNAPSHOT）;
end;

procedure TForm1.FormDestroy（Sender: TObject）;
begin
   //解除热键
   UnRegisterHotKey （Form1.Handle, id_SnapShot）;
end;

   //register   unregister
end.
```

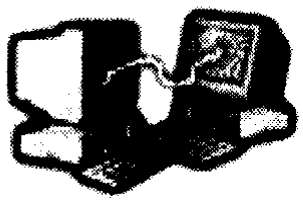
第十三章

API技术

热点透视

Delphi 功能之强已经是每一个程序员都认可的，但光凭借 Delphi 的 VCL 控件还远远不能充分发挥 Delphi 的功能，特别是对于底层的应用来说更是如此。由于 Windows API 的内容非常丰富，所以就凭短短的几十页纸是无法把其一一介绍完的，因此，本章只讲述编程过程中经常碰到的几个问题。本章内容包括：

- 防止加载应用程序的多个例程
- 获得另一程序的窗体句柄
- 列举指定窗体的子窗体
- 将应用程序图标放到托盘中
- 创建应用程序工具栏



防止加载应用程序的多个例程



遇到难题

对于许多应用程序来说，多个例程同时启动是毫无意义的。例如，对于字典类软件，一般在系统中启动字典例程就足够了，此时，需要确保整个系统中只有一个例程在运行，金山词霸就是如此。如何确保应用程序中只有一个例程在运行呢？如何尽可能恢复前一个已经运行的例程而关闭本次例程呢？



钥匙在此

在 Win16 中，当启动应用程序时，可以根据 hPrevInst 值来判断应用程序中例程的运行数目。如果该值非 0，表明应用程序中已经有一个实例在运行，并且可以废弃当前应用实例。在 Win32 中，hPrevInst 总是为 0，因此，不能依靠该函数来获取了。此时可以通过创建一个独一无二命名的互斥对象来实现。如果创建失败，则表明已经有另一个实例正在运行。在 Win32 API 中，CreateMutex（）函数用于创建已命名的或者未命名的对象。

CreateMutex（）函数在 windows.pas 中是这样声明的：

```
function CreateMutex（lpMutexAttributes: PSecurityAttributes; bInitialOwner: BOOL; lpName: PChar）: THandle; stdcall;
```

其中，参数 lpMutexAttributes 为 SECURITY_ATTRIBUTES 结构变量指针，决定返回的句柄能否被子线程所继承，如果该参数为空，则表明该句柄无法被继承使用。BInitialOwner 确定了互斥对象的初始化拥有者，如果为真，调用的线程将请求互斥对象的拥有关系。否则，互斥对象不被拥有。

lpName 指定了互斥对象的名字。在该名字中，除了反斜杠（\）可以拥有任何字符。该名字在对比时对大小写敏感。如果 lpName 与所存在的命名互斥对象名相匹配的话，这个函数则请求 MUTEX_ALL_ACCESS 来访问所存在的对象。如果 lpName 为空，表示该互斥对象不存在从而被创建。如果该名字与所存在的事件、文件映射对象等名字匹配时，该函数将发生错误。

如果该函数执行成功，返回值则是互斥对象的句柄。如果命名互斥对象在调用该函数时已经存在的话，则 GetLastError 函数将返回 ERROR_ALREADY_EXISTS，否则返回为 0。



跟我来

下例将创建命名互斥对象。如果互斥失败，则独一无二系统定制的窗体消息被系统所注册，并且广播给所有应用程序的顶层窗体，然后关闭第二个实例。先前的实例已经登记了这个独一无二的系统消息，并且可以截获响应这个消息的 Form1 窗体过程。如果消息得到了，Form1 将被恢复（如果被极小化了），然后将其显示为最上层的窗体。整个过程见清单 13-1。

清单 13-1　防止加载应用程序的多个例程

```
//项目文件
program oninstanced;

uses
   windows,
   Forms,
   oneinstance in 'oneinstance.pas' {Form1};

{$R *.RES}

begin
   {企图创建一个名为 MyApp 命名互斥体}
   CreateMutex（Nil, false, 'MyApp'）;
  {如果失败则表明另一个实例已经存在}
   if GetLastError = ERROR_ALREADY_EXISTS then begin
    {向自定义这些消息的所有窗体广播消息 }
    {仅仅自己的其他实例可以响应它，并将自己恢复过来}
     SendMessage（HWND_BROADCAST,
                  RegisterWindowMessage（'MyApp'）,//定义了独一无二系统消息
                  0, 0）;
     {退出}
     Halt（0）;
   end;
   Application.Initialize;
   Application.ShowMainForm := true;
   ShowWindow（Application.Handle, SW_RESTORE）;
   Application.CreateForm（TForm1, Form1）;
   Application.Run;
end.
```

```
//主窗体单元文件
unit oneinstance;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class（TForm）
    procedure FormCreate（Sender: TObject）;
    procedure FormDestroy（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

var
  OldWindowProc : Pointer; {老窗体过程变量}
  MyMsg : DWord; {定制系统消息}

function NewWindowProc（WindowHandle : hWnd;
                         TheMessage    : LongInt;
                         ParamW        : LongInt;
                         ParamL        : LongInt）  : LongInt stdcall;
begin
  if TheMessage = MyMsg  then begin
   {表明应用程序已被恢复并让其恢复窗体}
    SendMessage（Application.handle, WM_SYSCOMMAND, SC_RESTORE, 0）;
    SetForegroundWindow（Application.Handle）;
     BringWindowToTop（Form1.Handle）;
```

```
      setactivewindow（Form1.handle）;
     {已经处理完了这个消息}
     Result := 0;
     exit;
    end;
  {调用先前的窗体过程}
  Result := CallWindowProc（OldWindowProc,
                           WindowHandle,
                           TheMessage,
                           ParamW,
                           ParamL）;
end;

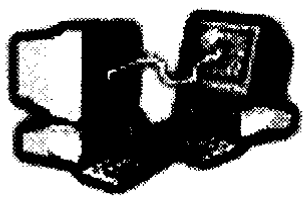
procedure TForm1.FormCreate（Sender: TObject）;
begin
 {登记定制的系统消息}
 MyMsg := RegisterWindowMessage（'MyApp'）;
 {设置窗体过程，并将老的窗体过程存储起来}
  OldWindowProc := Pointer（SetWindowLong（Form1.Handle,
                                          GWL_WNDPROC,
                                          LongInt（@NewWindowProc））;
end;

procedure TForm1.FormDestroy（Sender: TObject）;
begin
 {将 Form1 的窗体过程设置为原先的过程}
  SetWindowLong（Form1.Handle,
                GWL_WNDPROC,
                LongInt（OldWindowProc））;
end;

end.
```

在该程序的工程文件中，先试图创建一个名为 MyApp 的命名互斥体，如果创建失败表明该程序的例程已经存在。如果该应用程序已经有实例启动了，则向系统广播自定义的系统消息，然后结束本例程。

如果应用程序的实例先前并不存在，则继续执行程序。在应用程序的主窗体中，当该窗体创建时将登记定制的系统消息，并重新定义 Form1 的窗体过程，使其能响应系统发送的自定义消息，并告诉应用程序让其恢复窗体。



获得另一程序的窗体句柄



遇到难题

为了能够控制另一个应用程序或者获取另一个应用程序中的数据信息，除了通过消息机制来进行的方式之外，还可以通过获得该应用程序的窗体句柄来实现。例如，当要关闭操作系统时，则系统会向所有已运行的应用程序发出关闭程序的消息，如果该程序允许现在退出的话，程序自然给关闭了。另外，有许多解密软件可以直接获取指定编辑框中的数据，如当您的拨号上网的拨号器软件中存有密码时，往往可以通过这样的程序取出来，所以，如果将您的账号密码存储在系统中往往是不安全的。

有时，应用程序需要控制已运行的某一类应用程序。例如，当应用程序需要启动资源管理器时，如果该程序已经运行了，则将该程序显示在最外层，如果现在还未运行的话，则启动该应用程序。此时，则需要通过获得该程序的窗口句柄来实现。



钥匙在此

使用 Windows API 函数 FindWindow（）通过要定位的窗体标题或者类名可以非常简单直接地查找到该窗体。但一个应用程序的窗体类名往往是不知道的，并且有时又只知道这些窗体标题的一部分信息，此时就需要使用回调技术以及 EnumWindows（）函数来枚举所有活动的窗体，然后调用 Windows API 函数 GetWindowsText（）和 GetClassName（）来判断该窗体是否是所需要查找的窗体。当获得该窗体的句柄后可以随心所欲地控制该窗体了，例如，将该窗体标题栏写成自己的标题，或者关闭该应用程序，或者向该窗体中的编辑框发送数据等等。

回调函数是这样一个函数，它们不是由应用程序调用的，而是由 Windows 系统函数来调用的。即被系统函数调用的函数称之为回调函数。回调函数是作为系统函数的参数而被调用的。例如，EnumWindows（）函数就是这样的系统函数，它会将所有活动的顶层窗口的句柄作为参数传递给回调函数，由回调函数进行处理，直到所有窗口都列举完或者回调函数返回 False 为止。回调函数是根据系统传递给它的参数而进行相应处理的函数，这种函数往往是我们自己定义的。

为了获取指定窗体的句柄，需要调用 Windows API 系统函数 EnumWindows（）。该函数在 windows.pas 单元中是这样声明的：

```
function EnumWindows（lpEnumFunc: TFNWndEnumProc;
```

lParam: LPARAM）: BOOL; stdcall;

该函数枚举所有活动的顶层窗口的句柄，并将之作为参数传递给回调函数。该函数一直进行枚举，直到所有窗口都列举完或者回调函数返回 False 为止。

其中，参数 lpEnumFunc 即为应用程序所定义的回调函数的指针。所定义的函数的类型必须符合调用该回调函数的系统函数的要求。EnumWindows（）系统函数要求回调函数的类型应当与以下函数说明相同。

```
BOOL CALLBACK EnumWindowsProc（
    HWND hwnd, // 窗体句柄
    LPARAM lParam   //应用程序值
  ）;
```

如果回调函数中已经查找到所需要的窗体时，则该回调函数将返回 False，这时系统函数 EnumWindows（）就会停止继续枚举。

参数 lParam 是传递给回调函数的由应用程序所定义的值。

返回值：如果函数执行成功，则返回非 0，否则，返回值为 0。注意，该函数不会列举子窗体。

下面举一个使用回调函数的例子。

```
//回调函数的定义
function EnumWindowsProc（hWindow : hWnd;
                          lParam   : LongInt）  : Bool
{$IFDEF Win32} stdcall; {$ELSE} ; export; {$ENDIF}
begin
    //对系统函数所传递的参数值进行处理
    //如果已经符合要求，则应当设置 result:=false，以使系统函数不再进行枚举
    …
end;

function FindAWindow（Caption : string;
                      ClassName : string）  : THandle;
var
  WindowInfo : TFindWindowStruct;
begin
  begin
    //根据要查找的标题或类名给 WindowInfo 赋值
    WindowInfo.Caption := Caption;
    WindowInfo.ClassName := ClassName;
    WindowInfo.WindowHandle := 0;
    //开始枚举所有活动窗体，来查找符合条件的窗体
    EnumWindows（@EnumWindowsProc, LongInt（@WindowInfo））;
```

```
    //返回要查找的窗体句柄
    FindAWindow := WindowInfo.WindowHandle;
  end;
end;
```



跟我来

现在看一个例子，查找所有标题中带有“浏览”字眼的应用程序的窗体，如果查找到这样的窗体，则将其显示到最前端，如果未找到，则提示未找到并启动资源管理器。

新建应用程序，然后在窗体中添加一按钮，并为该按钮添加单击事件过程。完整例子参见清单 13-2。

清单 13-2　查找符合条件的窗体

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  PFindWindowStruct = ^TFindWindowStruct;
  TFindWindowStruct = record
    Caption : string;
    ClassName : string;
    WindowHandle : THandle;
  end;

  TForm1 = class（TForm）
    Button1: TButton;
    procedure Button1Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.DFM}

//用于查找所需要查找的窗体的回调函数
function EnumWindowsProc（hWindow : hWnd;
                          lParam   : LongInt）  : Bool
{$IFDEF Win32} stdcall; {$ELSE} ; export; {$ENDIF}
var
  lpBuffer : PChar;
  WindowCaptionFound : bool;
  ClassNameFound : bool;

begin
  GetMem（lpBuffer, 255）;
  Result := True;
  WindowCaptionFound := False;
  ClassNameFound := False;

  try
    //getwindowText 拷贝指定窗体的标题或者内容至 buffer 中
    if GetWindowText（hWindow, lpBuffer, 255）  > 0 then
      //判断是否为自己要查找的窗体
      if Pos（PFindWindowStruct（lParam）.Caption, StrPas（lpBuffer））  > 0 then
      WindowCaptionFound := true;
if PFindWindowStruct（lParam）.ClassName = '' then
  ClassNameFound := True else
  //将指定窗体的类名拷贝至 buffer 中
    if GetClassName（hWindow, lpBuffer, 255）  > 0 then
      if Pos（PFindWindowStruct（lParam）.ClassName, StrPas（lpBuffer））
       > 0 then ClassNameFound := True;

//如果已经全部找到的话，则告诉调用函数停止继续枚举
if （WindowCaptionFound and ClassNameFound）  then begin
  PFindWindowStruct（lParam）.WindowHandle := hWindow;
  Result := False;
```

```
    end;

  finally
    //释放所开辟的空间
    FreeMem（lpBuffer, sizeof（lpBuffer^））；
  end;
end;

function FindAWindow（Caption : string;
                      ClassName : string）  : THandle;
var
  WindowInfo : TFindWindowStruct;

begin
  begin
    //根据要查找的标题或类名给 WindowInfo 赋值
    WindowInfo.Caption := Caption;
    WindowInfo.ClassName := ClassName;
    WindowInfo.WindowHandle := 0;
    //开始枚举所有活动窗体，用来查找符合条件的窗体
    EnumWindows（@EnumWindowsProc, LongInt（@WindowInfo））；
    //返回要查找的窗体句柄
    FindAWindow := WindowInfo.WindowHandle;
  end;
end;

procedure TForm1.Button1Click（Sender: TObject）；
var
  TheWindowHandle : THandle;
begin
  TheWindowHandle := FindAWindow（'浏览', ''）；
  if TheWindowHandle = 0 then
  begin
    ShowMessage（'Window Not Found!'）；
    //运行资源管理器
    winexec（'EXPLORER.EXE',SW_SHOW ）；
  end
  else
    BringWindowToTop（TheWindowHandle）;//将符合条件的窗体显示在最前端
    {当然，此处也可以进行其他控制，如关闭该程序，或者继续查找该窗体中的子
```

窗体信息等等}
 end;

 end.

运行该应用程序，然后单击按钮，程序就会将资源管理器显示在您的面前。在回调函数中，通过 Windows API 函数 GetWindowText（）以及 GetClassName（）来判断是否已经找到标题中带有“浏览”字眼的窗体。



再送礼包

在上述代码中，调用了两个 Windows API 函数 GetWindowText（）以及 GetClassName（）。

GetWindowText（）函数主要用于将指定窗体的标题拷贝至缓冲区中，如果是编辑框这样的窗体的话，则拷贝的是该窗体中的内容。它带有三个参数，第一个参数为指定窗体的句柄；第二个参数为存放标题的缓冲区变量；第三个参数则为该缓冲区的大小。

GetClassName（）函数主要用于将指定窗体的类名拷贝至缓冲区中。其参数与 GetWindowText（）函数的参数基本类似。



列举指定窗体的子窗体



遇到难题

在上一节中，已经实现了获取指定窗体的句柄。但有时还需要对该窗体中的子窗体进行进一步的处理，此时，利用上一节的内容已经不能解决问题了，因为通过 EnumWindows（）函数只能得到最顶层窗体的句柄，而得不到该窗体下子窗体的句柄。例如，为了获取该窗体下所有标签的标题或者对其设置字体等等。当获得了某一窗体下所有子窗体的句柄后，就可以对这些子窗体进行处理了。



钥匙在此

当获得指定的主窗体句柄后，可以继续在该主窗体下列举所有的子窗体或者继续查找。

为了能枚举某一窗体下的子窗体，需要使用支持回调函数的一个系统函数EnumChildWindows（），该函数在 windows.pas 单元中是这样进行声明的：

```
function EnumChildWindows（hWndParent: HWND; lpEnumFunc: TFNWndEnumProc;
  lParam: LPARAM）: BOOL; stdcall;
```

该函数枚举指定窗体的所有子窗体，并将子窗体的窗口句柄作为参数传递给回调函数。该函数一直枚举直到所有子窗体枚举完或者回调函数返回 False 为止。

hWndParent 参数为指定父窗体的句柄。

lpEnumFunc 参数为回调函数的地址指针。该回调函数的类型格式应当与下列说明一致：

```
BOOL CALLBACK EnumChildProc（
  HWND hwnd, // 子窗体句柄
  LPARAM lParam   // 应用程序定义的值
）;
```

lParam 参数为应用程序自己定义的值。

下面举一个 EnumChildWindows（）调用回调函数的例子，该例子用于枚举指定窗体的所有子窗体的字体。

```
function EnumChildProc（HWnd: Integer; lParam: longint）: Boolean;
{$IFDEF Win32} stdcall; {$ELSE} ; export; {$ENDIF}
var
    ClsName,Caption   : array[0..31] of char;
    ClsLen     : Integer;
begin
    GetClassName（HWnd, ClsName, 31）;
    GetWindowtext（HWnd, Caption, 31）;
    //为指定的窗体设置字体
    SendMessage（HWnd, wm_SetFont, LoWord（lParam）, MAKELPARAM（1, 1））;
    //将类名以及标题值添加至 memo 中
    form1.Memo1.Lines.Add（'类名：'+clsname+'；    标题值：'+caption）;
    result:=true;
end;

//设置指定窗体下所有子窗体的字体
procedure SetWindowfont（awnd:THandle;Font:TFont）;
begin
      EnumChildWindows（awnd, @EnumChildProc, Font.Handle）;
end;
```



跟我来

现在看一个例子，它用了上一节的部分代码。该例子将 Windows 的拨号器主窗体下的所有子窗体（如按钮、标签、编辑框等）的字体设置成用户自己选择的字体，并且将所有子窗体的类名以及标题值显示出来。

在开始之前，先创建一个应用程序，并将一个 Tmemo 以及 Tbutton 控件放置在窗体上。该例子完整代码请参见清单 13-3。

清单 13-3　改变指定窗体下的所有子窗体的字体

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  PFindWindowStruct = ^TFindWindowStruct;
  TFindWindowStruct = record
    Caption : string;
    ClassName : string;
    WindowHandle : THandle;
  end;

  TForm1 = class（TForm）
    Button1: TButton;
    Memo1: TMemo;
    FontDialog1: TFontDialog;
    procedure Button1Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

```
implementation

{$R *.DFM}
//用于查找所需要查找的窗体的回调函数
function EnumWindowsProc（hWindow : hWnd;
                          lParam   : LongInt）  : Bool
{$IFDEF Win32} stdcall; {$ELSE} ; export; {$ENDIF}
var
  lpBuffer : PChar;
  WindowCaptionFound : bool;
  ClassNameFound : bool;
begin
  GetMem（lpBuffer, 255）;
  Result := True;
  WindowCaptionFound := False;
  ClassNameFound := False;

  try
    //getwindowText 拷贝指定窗体的标题或者内容至 buffer 中
    if GetWindowText（hWindow, lpBuffer, 255）  > 0 then
      //判断是否为自己要查找的窗体
      if Pos（PFindWindowStruct（lParam）.Caption, StrPas（lpBuffer））  > 0 then
      WindowCaptionFound := true;

    if PFindWindowStruct（lParam）.ClassName = '' then
      ClassNameFound := True else
      //将指定窗体的类名拷贝至 buffer 中
      if GetClassName（hWindow, lpBuffer, 255）  > 0 then
        if Pos（PFindWindowStruct（lParam）.ClassName, StrPas（lpBuffer））
         > 0 then ClassNameFound := True;

    //如果已经全部找到的话，则告诉调用函数停止继续枚举
    if （WindowCaptionFound and ClassNameFound）  then begin
      PFindWindowStruct（lParam）.WindowHandle := hWindow;
      Result := False;
    end;

  finally
    //释放所开辟的空间
```

```
      FreeMem（lpBuffer, sizeof（lpBuffer^））;
    end;
  end;

  function FindAWindow（Caption : string;
                        ClassName : string） : THandle;
  var
    WindowInfo : TFindWindowStruct;

  begin
    begin
      //根据要查找的标题或类名给 WindowInfo 赋值
      WindowInfo.Caption := Caption;
      WindowInfo.ClassName := ClassName;
      WindowInfo.WindowHandle := 0;
      //开始枚举所有活动窗体，用来查找符合条件的窗体
      EnumWindows（@EnumWindowsProc, LongInt（@WindowInfo））;
      //返回要查找的窗体句柄
      FindAWindow := WindowInfo.WindowHandle;
    end;
  end;

  function EnumChildProc（HWnd: Integer; lParam: longint）: Boolean;
  {$IFDEF Win32} stdcall; {$ELSE} ; export; {$ENDIF}
  var
     ClsName,Caption   : array[0..31] of char;
     ClsLen     : Integer;
  begin
     GetClassName（HWnd, ClsName, 31）;
     GetWindowtext（HWnd, Caption, 31）;
     //为指定的窗体设置字体
     SendMessage（HWnd, wm_SetFont, LoWord（lParam）, MAKELPARAM（1, 1））;
     //将类名以及标题值添加至 memo 中
     form1.Memo1.Lines.Add（'类名：'+clsname+'；    标题值：'+caption）;
     result:=true;
  end;

  //设置指定窗体下所有子窗体的字体
  procedure SetWindowfont（awnd:THandle;Font:TFont）;
  begin
```

```
        EnumChildWindows（awnd, @EnumChildProc, Font.Handle）;
end;

procedure TForm1.Button1Click（Sender: TObject）;
var
  TheWindowHandle : THandle;
begin
  //查找拨号器
  TheWindowHandle := FindAWindow（'连接到', ''）;
  if TheWindowHandle <> 0 then
  begin
      memo1.lines.clear;
      //打开字体设置对话框
      FontDialog1.Execute;
      //为指定窗体的所有子窗体设置字体
      setWindowfont（TheWindowHandle, FontDialog1.Font）;
   end
end;
end.
```

该代码中，先由上一节的例子实现了主窗体的查找，本例中是查找拨号网络中的拨号连接器。当查找到该窗体之后，根据用户所选择的字体，由 SetWindowfont（）主函数设置拨号连接器所有标签、按钮以及编码框的字体。

为了看一下该代码的运行结果，请先运行拨号器，然后运行该程序，并单击按钮。此时，拨号连接器的字体将发生改变。同时，应用程序中的 Memo 控件将把所有子窗体的类名以及标题值显示出来。连接器的字体发生的变化请参见图 13-1。显示子窗体的类名以及标题值的图请参见图 13-2。通过本例希望能起到抛砖引玉的作用。

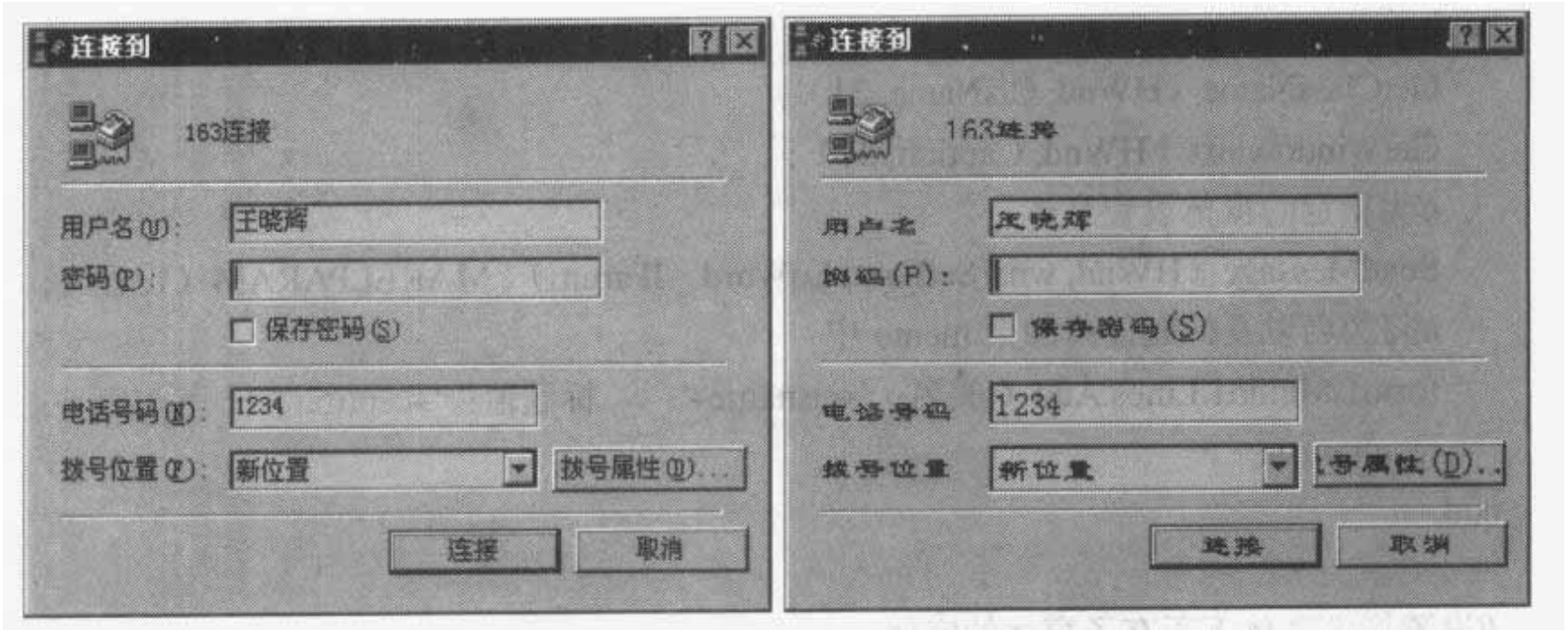


图 13-1　连接器的字体发生了变化

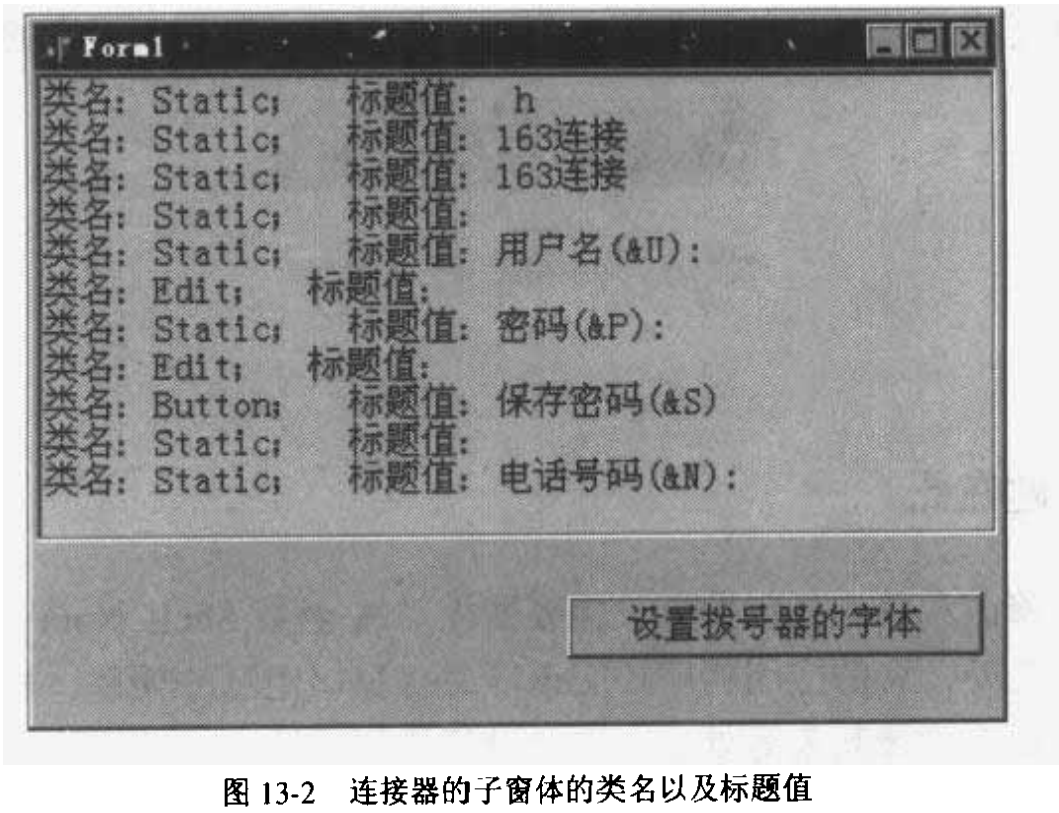


图 13-2　连接器的子窗体的类名以及标题值



再送礼包

为了能对指定窗体设置字体，本节使用的是发送 WM_SETFONT 消息的方式，这是因为考虑到指定窗体下的子窗体的种类不一样。

WM_SETFONT 消息类型中的参数解释如下：

wParam = （WPARAM） hfont;　　// 字体句柄

lParam = MAKELPARAM（fRedraw, 0）;　　// 重画标志

参数 wParam 指定的是字体的句柄；参数 lParam 为重画标志，指定在设置字体时是否立即更改指定窗体的字体，当 fRedraw 为真时，表示立即更新字体。

对于该消息的使用请参见下列语句：

SendMessage（HWnd, wm_SetFont, FontHandle, MAKELPARAM（1, 1））;



将应用程序图标放到托盘中



遇到难题

在 Windows 的桌面中，一些服务程序或者系统监测程序往往将其图标放置在任务栏的右下角，这样有利于程序快捷地启动或者配置。例如，我的 Windows 的任务栏中的托盘小图标显示如图 13-5 所示。在该托盘中的图标分别代表了任务管理程序、创通的 AudioPCI 混音器、Norton AntiVirus 杀毒软件、SQLServer7.0 服务启动程序、金山词霸、输入法程序以

及时钟程序。也许您也想将自己的程序放入到这个托盘中，以利于用户的启动或关闭。



图 13-3　我的任务栏中的托盘图标



钥匙在此

其实，创建、修改和移走托盘图标只需要调用 API 函数 Shell_NotifyIcon（）即可完成。但除了调用该函数之外，您的应用程序还应该能进行窗口消息的处理、显示图标和提示、鼠标点击以及将应用程序隐藏起来等处理。下面进行详细阐述。

1. shell_NotifyIcon（）函数

该函数在 shellAPI.pas 单元文件中是这样进行声明的：

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData: PNotifyIconData): BOOL; stdcall;
```

dwMessage 参数描述了要对图标进行的操作，它可以为表 13-1 中的任意一个值。

表 13-1　　dwMessage 参数取值

常数	值	含 义
NIM_ADD	0	将一个图标加到托盘中
NIM_DELETE	1	将一个图标从托盘中移走
NIM_MODIFY	2	修改一个图标数据

lpData 是指向 TNOTIFYICONDATA 记录的指针。这个记录是这样进行声明的：

```
type
TNOTIFYICONDATA = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..63] of AnsiChar;
  end;
```

其中，cbsize 域是这个记录的大小，应当初始化为 sizeof（TNOTIFYICONDATA）。

Wnd 域是窗口句柄，托盘图标的回调消息就是发送给这个窗口句柄的。

uID 域是程序员定义的唯一标识符。如果一个应用程序有多个托盘图标的话，则每一个图标都有一个唯一的 ID。

uFlags 域描述了 TNOTIFYICONDATA 记录中哪个域是活动的，它可以设为表 13-2 中所列的某个值。

表 13-2　　uFlags 取值

常量	值	含义
NIF_ICON	0	HIcon 域是有效的
NIF_MESSAGE	1	UCallbackMessage 域是有效的
NIF_TIP	2	SzTip 域是有效的

UCallbackMessage 域就是发向由 Wnd 字段标识的窗口的消息。一般来说，这个域的值可以通过 RegisterWindowMessage（）来获得，也可以通过 WM_User 的偏移量获得。这个消息的 Iparam 参数值与 uID 域的值相同，wParam 参数表示鼠标经过图标上时发送的消息。

HIcon 域就是放到托盘上的图标的句柄。

Sztip 域是一个字符串，当鼠标指向图标时，会弹出一个提示框显示这个字符串。

例如，下列例子说明了如何在托盘上添加图标。

```
const
    IconData : TNotifyIconData;
begin
  //设置 TNotifyIconData 变量值
  IconCount := 0;
  IconData.cbSize := sizeof（IconData）;
  IconData.Wnd := Handle;
  IconData.uID := 100;
  IconData.uFlags := NIF_MESSAGE + NIF_ICON + NIF_TIP;
  IconData.uCallbackMessage := WM_USER + 1;
  IconData.hIcon := Application.Icon.Handle;
  //以应用程序的标题作为图标提示
  StrPCopy（IconData.szTip, Application.Title）;
  //向托盘中添加图标
  Shell_NotifyIcon（NIM_ADD, @IconData）;
End;
```

在该例子中，先对 TnotifyIconData 变量进行数据的赋值，然后调用 Shell_NotifyIcon（）函数向托盘中添加图标。

2. 点击托盘图标的消息处理

当鼠标单击图标时，为了能对这一操作进行响应，需要在点击图标时进行相应消息处理。例如，当鼠标右击图标时应当弹出一个菜单或者启动应用程序等等。为了能响应鼠标的单击事件，需要对窗口处理过程进行重载。

例如，假设一应用程序的窗体名为 Form1，则下列代码重载了窗口过程，使其能截获鼠标的事件。

```
TForm1 = class（TForm）
...
    procedure WndProc（var Msg : TMessage）; override;
```

```
...
end;
...
procedure TForm1.WndProc（var Msg : TMessage）;
var
  p : TPoint;
begin
  case Msg.Msg of
    WM_USER + 1:
    //在鼠标点击托盘图标时进行处理
    ...
    end;//end of case
    end;
    inherited;
end;
```

3. 托盘图标及提示

要将图标及提示显示给用户很简单，只要设置 TNOTIFYICONDATA 结构变量中的 hIcon 以及 szTip 域值即可。如果您希望图标和提示都能动态改变的话，可以在某事件发生时，重新修改这两个域值，同时调用 Shell_NotifyIcon（）进行更改图标数据。这样就可以实现动态地显示图标和提示了。

例如，下例代码演示了当时间发生变化时，如何更改图标和显示内容。假设窗体和 TNOTIFYICONDATA 结构变量与上述例子中的一致，则代码如下：

```
procedure TForm1.Timer1Timer（Sender: TObject）;
begin
  //随着时间的变化动态地更改图标
  case（IconCount） of
    0 : IconData.hIcon := LoadIcon（0, IDI_APPLICATION）;
    1 : IconData.hIcon := LoadIcon（0, IDI_ASTERISK）;
    2 : IconData.hIcon := LoadIcon（0, IDI_EXCLAMATION）;
    3 : IconData.hIcon := LoadIcon（0, IDI_HAND）;
    4 : IconData.hIcon := LoadIcon（0, IDI_QUESTION）;
    5 : IconData.hIcon := Application.Icon.Handle;
  end;
  inc（IconCount）;
  if IconCount > 5 then
    IconCount := 0;
  //将当前的时间作为应用程序的标题来显示
  Application.Title := TimeToStr（Now）;
  StrPCopy（IconData.szTip, Application.Title）;
  //更新托盘图标数据
```

```
  Shell_NotifyIcon（NIM_MODIFY, @IconData）;
end;
```

4. 鼠标点击处理

上述 2 中已经讲述了如何对单击图标事件进行处理，但应当对不同的鼠标操作进行不同的处理，如对鼠标左右键的不同处理等等。例如当鼠标右击图标时应当弹出菜单进行选择，当双击图标时，应当显示应用程序等等。对于是鼠标左键还是右键的信息存储在消息参数中。该消息的 lParam 参数值可以为 WM_RBUTTONDOWN、WM_LBUTTONDOWN 和 WM_LBUTTONDBCLICK 等，分别表示右键单击、左键单击和左键双击等。

本例将在鼠标右键点击图标时弹出对话框，代码如下：

```
    case Msg.lParam of
      //如果是鼠标右键，则在鼠标位置弹出菜单
      WM_RBUTTONDOWN: begin
        GetCursorPos（p）;
        PopupMenu1.Popup（p.x, p.y）;
    end //end of case
```

5. 隐藏应用程序

为了使具有托盘图标功能的应用程序不在 Windows 的任务栏上显示为一个按钮，需要将应用程序隐藏起来。要隐藏应用程序，可以调用 ShowWindows（）函数。由这个函数传递给应用程序的句柄 handle 以及一个常量来决定是显示还是隐藏应用程序的图标。

例如，要使应用程序隐藏起来，可以调用以下代码：

```
  ShowWindow（Application.Handle, SW_HIDE）;
```

假如当鼠标双击图标时需要显示应用程序并弹出应用程序的主窗体，可以使用以下代码：

```
begin
  //显示窗体
  Form1.Show;
  //在任务栏中隐藏应用程序
  ShowWindow（Application.Handle, SW_HIDE）;
end;
```



跟我来

下列例子显示了如何在系统的托盘中显示不同的图标和标题。该应用程序将会响应鼠标的右键，并弹出菜单供用户选择是否显示窗体，并且当窗体显示出来后，在任务栏中就不出现该应用程序，即在任务栏中隐藏了该应用程序。

为了完成该应用程序，需要创建一个载有 Tpopupmenu 以及 Ttimer 的控件，并且需要修改工程文件。清单 13-4 列出了本例的完整代码。

清单 13-4　托盘图标应用程序

```
{TrayIt.dpr}

program TrayIt;

uses
  Windows,
  Forms,
  TrayIt1 in 'TrayIt1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.ShowMainForm := False;//不显示主窗体
  Application.CreateForm（TForm1, Form1）;
  ShowWindow（Application.Handle, SW_HIDE）;  //在任务栏中隐藏应用程序
  Application.Run;
end.

{TrayIt1.pas}

unit TrayIt1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Menus, ShellAPI, ExtCtrls;

type
  TForm1 = class（TForm）
    PopupMenu1: TPopupMenu;
    Open1: TMenuItem;
    Exit1: TMenuItem;
    Timer1: TTimer;
    procedure FormCreate（Sender: TObject）;
    procedure Open1Click（Sender: TObject）;
    procedure Exit1Click（Sender: TObject）;
```

```
    procedure FormClose（Sender: TObject; var Action: TCloseAction）;
    procedure Timer1Timer（Sender: TObject）;
  private
    { Private declarations }
    procedure WndProc（var Msg : TMessage）; override;
  public
    { Public declarations }
    IconData : TNotifyIconData;
    IconCount : integer;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
//重载窗口处理函数，以截取自定义消息
procedure TForm1.WndProc（var Msg : TMessage）;
var
  p : TPoint;
begin
  case Msg.Msg of
    WM_USER + 1:
    //在鼠标点击托盘图标时进行处理
    case Msg.lParam of
      //如果是鼠标右键，则在鼠标位置弹出菜单
      WM_RBUTTONDOWN: begin
        GetCursorPos（p）;
        PopupMenu1.Popup（p.x, p.y）;
      end //end of case
    end;//end of case
  end;
  inherited;
end;

//初始化窗体
procedure TForm1.FormCreate（Sender: TObject）;
begin
  BorderIcons := [biSystemMenu];
```

```
    //设置 TNotifyIconData 变量值
    IconCount := 0;
    IconData.cbSize := sizeof（IconData）;
    IconData.Wnd := Handle;
    IconData.uID := 100;
    IconData.uFlags := NIF_MESSAGE + NIF_ICON + NIF_TIP;
    IconData.uCallbackMessage := WM_USER + 1;
    IconData.hIcon := Application.Icon.Handle;
    //以应用程序的标题作为图标提示
    StrPCopy（IconData.szTip, Application.Title）;
    //向托盘中添加图标
    Shell_NotifyIcon（NIM_ADD, @IconData）;
    //启动时钟事件
    Timer1.Interval := 1000;
    Timer1.Enabled := true;
end;

procedure TForm1.Open1Click（Sender: TObject）;
begin
    //显示窗体
    Form1.Show;
    //在任务栏中隐藏应用程序
    ShowWindow（Application.Handle, SW_HIDE）;
end;

procedure TForm1.Exit1Click（Sender: TObject）;
begin
    //删除托盘图标
    Shell_NotifyIcon（NIM_DELETE, @IconData）;
    Application.ProcessMessages;
    Application.Terminate;
end;

procedure TForm1.FormClose（Sender: TObject; var Action: TCloseAction）;
begin
    //关闭显示的窗体
    Action := caNone;
    Form1.Hide;
end;
```

```
procedure TForm1.Timer1Timer（Sender: TObject）;
begin
  //随着时间的变化动态地更改图标
  case（IconCount） of
    0 : IconData.hIcon := LoadIcon（0, IDI_APPLICATION）;
    1 : IconData.hIcon := LoadIcon（0, IDI_ASTERISK）;
    2 : IconData.hIcon := LoadIcon（0, IDI_EXCLAMATION）;
    3 : IconData.hIcon := LoadIcon（0, IDI_HAND）;
    4 : IconData.hIcon := LoadIcon（0, IDI_QUESTION）;
    5 : IconData.hIcon := Application.Icon.Handle;
  end;
  inc（IconCount）;
  if IconCount > 5 then
    IconCount := 0;
  //将当前的时间作为应用程序的标题来显示
  Application.Title := TimeToStr（Now）;
  StrPCopy（IconData.szTip, Application.Title）;
  //更新托盘图标数据
  Shell_NotifyIcon（NIM_MODIFY, @IconData）;
end;

begin
  //在任务栏中隐藏应用程序
  ShowWindow（Application.Handle, SW_HIDE）;
end.
```

运行该程序，结果会怎样呢？在托盘中多了一个图标，并且还不断在变化，那其实就是应用程序的托盘图标，结果如图 13-4 所示，比图 13-3 自然多了一个图标，由于程序中实现了动态图标的更改，其中图（a）、（b）分别为两个不同时刻下的结果。

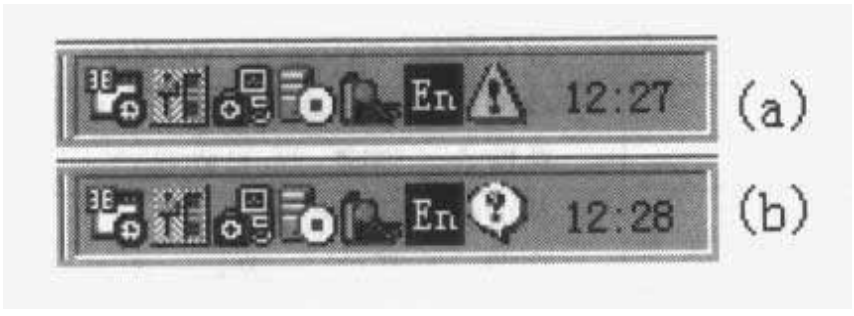


图 13-4　应用程序的托盘图标



创建应用程序工具栏



遇到难题

我们已经对系统的任务栏再熟悉不过了，当您启动了 Win 95/98 或者 NT4.0 都可以见到它，一般情况下它停留在桌面的最下边，除非您已经将其拖到其他的边缘上。在系统任务栏上放置了一个开始按钮、托盘图标以及已经启动的应用程序的标题等等。

任务栏的一个特点是，它可以在桌面的边缘上进行停靠，并且可以自动隐藏、拖放等。同样，对于您自己的应用系统，也可以编写类似功能的程序。这就是应用程序工具栏，以下一般称 AppBar。



钥匙在此

您也许会吃惊，完成这些功能只需要一个函数。就像托盘图标一样，设计 AppBar 只需要一个 API 函数，即 SHAppBarMessage（）函数。这个函数在 shellAPI.pas 单元文件中是这样进行声明的：

```
function SHAppBarMessage（dwMessage: DWORD;
var pData: TAppBarData）: UINT; stdcall;
```

该函数用于将 AppBar 消息发送到系统中。

参数 dwMessage 为发送的消息标识符。该参数可以取表 13-3 中值之一。

表 13-3　　dwMessage 取值

值	含 义
ABM_ACTIVATE	通知系统，AppBar 已被激活
ABM_GETAUTOHIDEBAR	获取停靠在屏幕边缘且自动隐藏的 AppBar 的句柄
ABM_GETSTATE	获取自动隐藏、总是在最前的状态
ABM_GETTASKBARPOS	获取任务栏的位置
ABM_NEW	注册一个新的 AppBar，指定一个新的回调消息
ABM_QUERYPOS	请求新的位置和尺寸
ABM_REMOVE	撤销一个 AppBar 的注册
ABM_SETAUTOHIDEBAR	注册一个自动隐藏的 AppBar
ABM_SETPOS	设置一个新的位置和尺寸
ABM_WINDOWPOSCHANGED	通知外壳程序，AppBar 的位置已经改变

参数 pData 为一个 TAppBarData 记录变量指针，该结构的内容取决于 dwMessage 参数的值。

```
PAppBarData = ^TAppBarData;
  TAppBarData = record
    cbSize: DWORD;//变量的大小
    hWnd: HWND;//AppBar 的窗口句柄
    uCallbackMessage: UINT;//回调函数消息
    uEdge: UINT;//边缘类型
    rc: TRect;//矩形坐标
    lParam: LPARAM; { message specific }
  end;
```

这个记录中，cbSize 为该变量结构的大小；hWnd 域是 AppBar 的窗口句柄。

uCallbackMessage 域为应用程序定义的消息标识符。应用程序使用该指定的标识符以便将通知消息发送给 hWnd 域辨认。该成员在发送 ABM_NEW 消息时使用。

uEdge 域为停靠类型的标志，用于指定屏幕的边缘。该域可以取表 13-4 中值之一。

表 13-4　　uEdge 域取值

值	含 义
ABE_BOTTOM	下边缘
ABE_LEFT	左边缘
ABE_RIGHT	右边缘
ABE_TOP	上边缘

该域在发送 ABM_GETAUTOHIDEBAR 、ABM_QUERYPOS、ABM_SETAUTOHIDEBAR 和 ABM_SETPOS 消息时使用。

域 rc 为 AppBar 的边界矩形，该域在发送 ABM_GETTASKBARPOS、ABM_QUERYPOS 和 ABM_SETPOS 消息时使用。

LParam 域的含义取决于 dwMessage 消息，它与 ABM_SETAUTOHIDEBAR 消息一起使用。

说 明　在 Win32 的在线帮助的 AppBarData 以及 SHAppBarMessage（）函数中有更详细的介绍。



跟我来

清单 13-5 是一个创建应用程序工具栏 AppBar 的完整例子。

清单 13-5　应用程序工具栏 AppBar

```
unit AppBars;
```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ShellAPI, StdCtrls, Buttons, ExtCtrls, Menus, ImgList, ComCtrls, ToolWin;

type
  TAppBarEdge = （abeTop, abeBottom, abeLeft, abeRight）;

  EAppBarError = class（Exception）;

  TMyExeRec=record
    ExeName,
    Description:string;
end;

TAppBar = class（TForm）
  PopupMenu1: TPopupMenu;
  Top1: TMenuItem;
  Bottom1: TMenuItem;
  Left1: TMenuItem;
  Right1: TMenuItem;
  N1: TMenuItem;
  Exit1: TMenuItem;
  ToolBar1: TToolBar;
  ToolButton1: TToolButton;
  ToolButton2: TToolButton;
  ToolButton3: TToolButton;
  ToolButton4: TToolButton;
  ToolButton5: TToolButton;
  ImageList1: TImageList;
  procedure Top1Click（Sender: TObject）;
  procedure MoveButtons;
  procedure Exit1Click（Sender: TObject）;
  procedure FormShow（Sender: TObject）;
  procedure ToolButton1Click（Sender: TObject）;
private
  FABD: TAppBarData;
  FEdge: TAppBarEdge;
  FTopMost: Boolean;
```

```
    FLastChecked: TMenuItem;
    procedure WMActivate（var M: TMessage）; message WM_ACTIVATE;
    procedure WMWindowPosChanged（var M: TMessage）; message
WM_WINDOWPOSCHANGED;
    function SendAppBarMsg（Msg: DWORD）: UINT;
    procedure SetAppBarPos（Edge: UINT）;
    procedure SetAppBarEdge（Value: TAppBarEdge）;
  protected
    procedure CreateParams（var Params: TCreateParams）; override;
    procedure SetTopMost（TopMost: Boolean）;
    procedure WndProc（var M: TMessage）; override;
  public
    constructor Create（AOwner: TComponent）; override;
    destructor Destroy; override;
  published
    property Edge: TAppBarEdge read FEdge write SetAppBarEdge;
    property TopMost: Boolean read FTopMost write SetTopMost default True;
  end;

var
  AppBar: TAppBar;

implementation

{$R *.DFM}

const
  DEF_APPBAR_WIDTH  = 40;
  DEF_APPBAR_HEIGHT = 35;
  MyExe:array[0..4]of TMyExeRec=（
        （Exename:'EXPLORER.EXE';Description:'资源管理器'），
        （Exename:'NOTEPAD.EXE';Description:'记事本'），
        （Exename:'CALC.EXE';Description:'计算器'），
        （Exename:'CDPlayer.exe';Description:'CD 播放器'），
        （Exename:'MPlayer.exe';Description:'媒体播放器'））；

var
  AppBarMsg: UINT = 0;

constructor TAppBar.Create（AOwner: TComponent）;
```

```
begin
  inherited Create（AOwner）;
  // 设置 TAppBarData 记录
  with FABD do
  begin
    cbSize := SizeOf（FABD）;
    hWnd := Handle;
    uCallbackMessage := AppBarMsg;
  end;
  // 通知外壳程序，创建新应用程序工具条
  if SendAppBarMsg（ABM_NEW） = 0 then
    raise EAppBarError.Create（'Failed to create AppBar'）;
  // 初始化 AppBar 的位置，并将之放在最上方
  SetAppBarPos（ABE_Top）;
  FTopMost := True;
  FLastChecked := Top1;
end;

destructor TAppBar.Destroy;
begin
  // 通知外壳程序，该 AppBar 已经被移开了
  SendAppBarMsg（ABM_REMOVE）;
  inherited Destroy;
end;

procedure TAppBar.WMWindowPosChanged（var M: TMessage）;
begin
  inherited;
  // 通知外壳程序，AppBar 的位置已经被改变
  SendAppBarMsg（ABM_WINDOWPOSCHANGED）;
end;

procedure TAppBar.WMActivate（var M: TMessage）;
begin
  inherited;
  // 通知外壳程序，AppBar 窗口被激活
  SendAppBarMsg（ABM_ACTIVATE）;
end;

procedure TAppBar.WndProc（var M: TMessage）;
```

```
var
  State: UINT;
  WndPos: HWnd;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
      // 当在上端或者自动隐藏被改变时被发送
      ABN_STATECHANGE:
        begin
          // 检查工具条是否仍然是 ABS_ALWAYSONTOP
          State := SendAppBarMsg（ABM_GETSTATE）;
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost（False）
          else
            SetTopMost（True）;
        end;
      // 满屏应用程序已经启动，或者上一个满屏幕应用程序已经被关闭
      ABN_FULLSCREENAPP:
        begin
          // 设置工具条的 z-order
          State := SendAppBarMsg（ABM_GETSTATE）;
          if M.lParam <> 0 then begin
            if ABS_ALWAYSONTOP and State = 0 then
              SetTopMost（False）
            else
              SetTopMost（True）;
          end
          else
            if State and ABS_ALWAYSONTOP <> 0 then
              SetTopMost（True）;
        end;
      // 当影响 AppBar 的位置时被发送
      ABN_POSCHANGED:
        begin
          // 任务栏或者其他工具条的位置被改变了
          SetAppBarPos（FABD.uEdge）;
        end;
    end;
  end
```

```
    else
      inherited WndProc（M）;
end;

function TAppBar.SendAppBarMsg（Msg: DWORD）: UINT;
begin
  Result := SHAppBarMessage（Msg, FABD）;
end;

procedure TAppBar.SetAppBarPos（Edge: UINT）;
begin
  FABD.uEdge := Edge;            // set edage
  with FABD.rc do
  begin
    // 设置坐标为满屏
    Top := 0;
    Left := 0;
    Right := Screen.Width;
    Bottom := Screen.Height;
    //发送 ABM_QUERYPOS 消息以获得适当的矩形边缘
    SendAppBarMsg（ABM_QUERYPOS）;
    // 根据被修改的位置重新调整矩形
    case Edge of
      ABE_LEFT: Right := Left + DEF_APPBAR_WIDTH;
      ABE_RIGHT: Left := Right - DEF_APPBAR_WIDTH;
      ABE_TOP: Bottom := Top + DEF_APPBAR_HEIGHT;
      ABE_BOTTOM: Top := Bottom - DEF_APPBAR_HEIGHT;
    end;
    // 设置 AppBar 的位置
    SendAppBarMsg（ABM_SETPOS）;
  end;
  // 设置 BoundsRect 属性，以便将 bounding 矩形通知系统
  BoundsRect := FABD.rc;
end;

procedure TAppBar.SetTopMost（TopMost: Boolean）;
const
  WndPosArray: array[Boolean] of HWND =  （HWND_BOTTOM, HWND_TOPMOST）;
begin
  SetWindowPos（Handle, WndPosArray[TopMost], 0, 0, 0, 0, SWP_NOMOVE or
```

```
    SWP_NOSIZE or SWP_NOACTIVATE）;
end;

procedure TAppBar.CreateParams（var Params: TCreateParams）;
begin
  inherited CreateParams（Params）;
  Params.ExStyle := WS_EX_TOOLWINDOW or WS_EX_TOPMOST;
  Params.Style := WS_VISIBLE or WS_POPUP or
        WS_THICKFRAME or WS_CLIPCHILDREN;
end;

procedure TAppBar.SetAppBarEdge（Value: TAppBarEdge）;
const
  EdgeArray: array[TAppBarEdge] of UINT =  （ABE_TOP, ABE_BOTTOM, ABE_LEFT,
    ABE_RIGHT）;
begin
  if Value <> FEdge then
  begin
    SetAppBarPos（EdgeArray[Value]）;
    FEdge := Value;
    MoveButtons;
  end;
end;

procedure TAppBar.Top1Click（Sender: TObject）;
begin
  FLastChecked.Checked := False;
   （Sender as TMenuItem）.Checked := True;
  case TMenuItem（Sender）.Caption[2] of
    'T': Edge := abeTop;
    'B': Edge := abeBottom;
    'L': Edge := abeLeft;
    'R': Edge := abeRight;
  end;
  FLastChecked := TMenuItem（Sender）;
end;

procedure TAppBar.MoveButtons;
// 该方法虽然看上去比较复杂，但它是真正的依靠按钮的大小来使 AppBar 停靠的
var
```

```
    DeltaCenter, NewPos: Integer;
  begin
    if （FEdge = abeTop） or （FEdge = abeBottom） then
    begin
      DeltaCenter := （ClientHeight - toolButton1.Height） div 2;
      toolButton1.SetBounds（10, DeltaCenter, toolButton1.Width,
                              toolButton1.Height）;
      NewPos := toolButton1.Width + 20;
      toolButton2.SetBounds（NewPos, DeltaCenter, toolButton1.Width,
                              toolButton1.Height）;
      NewPos := NewPos + toolButton1.Width + 10;
      toolButton3.SetBounds（NewPos, DeltaCenter, toolButton1.Width,
                              toolButton1.Height）;
    NewPos := NewPos + toolButton1.Width + 10;
    toolButton4.SetBounds（NewPos, DeltaCenter, toolButton1.Width,
                              toolButton1.Height）;
    NewPos := NewPos + toolButton1.Width + 10;
    toolButton5.SetBounds（NewPos, DeltaCenter, toolButton1.Width,
                              toolButton1.Height）;
  end;
  if （FEdge = abeLeft） or （FEdge = abeRight） then
  begin
    DeltaCenter := （ClientWidth - toolButton1.Width） div 2;
    toolButton1.SetBounds（DeltaCenter, 10, toolButton1.Width,
                    toolButton1.Height）;
    NewPos := toolButton1.Height + 20;
    toolButton2.SetBounds（DeltaCenter, NewPos, toolButton1.Width,
                    toolButton1.Height）;
      NewPos := NewPos + toolButton1.Height + 10;
      toolButton3.SetBounds（DeltaCenter, NewPos, toolButton1.Width,
                    toolButton1.Height）;
      NewPos := NewPos + toolButton1.Height + 10;
      toolButton4.SetBounds（DeltaCenter, NewPos, toolButton1.Width,
                    toolButton1.Height）;
      NewPos := NewPos + toolButton1.Height + 10;
      toolButton5.SetBounds（DeltaCenter, NewPos, toolButton1.Width,
                    toolButton1.Height）;
    end;
  end;
```

```
procedure TAppBar.Exit1Click（Sender: TObject）;
begin
   Application.Terminate;
end;

procedure TAppBar.FormShow（Sender: TObject）;
var
   i:integer;
   MyIcon:TIcon;
   j:word;
begin
      //将各按钮的提示显示为应用程序的名称,图标显示为应用程序的图标
      imagelist1.Clear;
      for i:=0 to 4 do begin
          ToolBar1.Buttons[i].Hint:=myExe[i].Description;
          myicon:=ticon.Create;
          MyIcon.Handle:=ExtractAssociatedIcon（Handle,
                                    pchar（myExe[i].ExeName）,j）;
          if MyIcon.handle <>null then
           ImageList1.AddIcon（myIcon）;
      end;
end;

procedure TAppBar.ToolButton1Click（Sender: TObject）;
begin
      //该事件已被 toolbutton1~toolbutton5 所引用
      //根据不同的按钮执行连接的相应应用程序
winexec（pchar（myexe[TToolbutton（sender）.imageindex].exename）,SW_SHOW）;
      end;

      initialization
        //注册了 window 消息
        AppBarMsg := RegisterWindowMessage（'AppBar Message'）;
      end.
```

在该程序代码中，函数 SendAppBarMsg（）实际上给系统函数 SHAppBarMessage（）加上了一个外套。

```
      function TAppBar.SendAppBarMsg（Msg: DWORD）: UINT;
      begin
        Result := SHAppBarMessage（Msg, FABD）;
```

```
end;
```

在该窗体创建时，为应用程序注册创建了新工具栏 AppBar，并设置了 AppBar 的初始位置。当窗体 show 时，则将代表各应用程序的按钮进行 hint 以及图标的赋值，使其在显示时一目了然，知道哪个按钮代表哪个执行程序。

```
procedure TAppBar.FormShow（Sender: TObject）;
var
  i:integer;
  MyIcon:TIcon;
  j:word;
begin
    //将各按钮的提示显示为应用程序的名称,图标显示为应用程序的图标
    imagelist1.Clear;
    for i:=0 to 4 do begin
        ToolBar1.Buttons[i].Hint:=myExe[i].Description;
        myicon:=ticon.Create;
        MyIcon.Handle:=ExtractAssociatedIcon（Handle,
                                    pchar（myExe[i].ExeName）,j）;
        if MyIcon.handle <>null then
         ImageList1.AddIcon（myIcon）;
    end;
end;
```

前面已经提到，AppBar 的通知消息将发送到 FABD.hwnd 所标识的窗体上，要发送的消息则由域 FABD.uCallBackMessage 表明。为了使应用程序的工具栏能够响应系统所发送的关于 AppBar 的消息，需要对窗体的窗口函数进行重载。

```
procedure TAppBar.WndProc（var M: TMessage）;
var
  State: UINT;
  WndPos: HWnd;
begin
  if M.Msg = AppBarMsg then
  begin
    case M.WParam of
      // 当总是在上端或者自动隐藏被改变时被发送
      ABN_STATECHANGE:
        begin
          // 检查工具条是否仍然是 ABS_ALWAYSONTOP
          State := SendAppBarMsg（ABM_GETSTATE）;
          if ABS_ALWAYSONTOP and State = 0 then
            SetTopMost（False）
```

```
      else
        SetTopMost（True）;
    end;
  // 满屏应用程序已经启动，或者上一个满屏幕应用程序已经被关闭
  ABN_FULLSCREENAPP:
    begin
      // 设置工具条的 z-order
      State := SendAppBarMsg（ABM_GETSTATE）;
      if M.lParam <> 0 then begin
        if ABS_ALWAYSONTOP and State = 0 then
          SetTopMost（False）
        else
          SetTopMost（True）;
        end
        else
          if State and ABS_ALWAYSONTOP <> 0 then
            SetTopMost（True）;
        end;
      // 当影响 AppBar 的位置时被发送
      ABN_POSCHANGED:
        begin
          // 任务栏或者其他工具条的位置被改变了
          SetAppBarPos（FABD.uEdge）;
        end;
    end;
end
else
  inherited WndProc（M）;
end;
```

运行该应用程序，则初始时该程序工具栏自动停靠在屏幕顶端，该窗体根据所给定的应用程序来获得相应的图标并显示到 ImageList 中，由于将 toolbar 的 Imagelist 选项设置成了 imagelist1，并且各 toolbutton 的 imageindex 已经进行了设置，所以，窗体上显示的图标都为正在执行程序的图标。在窗体上用鼠标右键单击应用程序工具栏，然后选择 bottom 则会将该 AppBar 停靠在最底端，如图 13-5 所示。当鼠标单击图标按钮时，将执行相应的应用程序。



图 13-5　应用程序工具栏

第十四章

图形与图像

热点透视

许多程序员觉得图形开发是编程中最令人愉快的事情。因为图形使得应用程序能将自己画在屏幕上，同时也是您的艺术天才得以发挥的工具。本章主要讲述了如何通过 TCanvas 以及 Windows GDI 来实现画图或对图像进行处理，同时也讲述了如何操纵 Windows Desktop 以及实现自己的字体等。本章内容主要有：

- ✧ 理解 WindowsGDI 与 TCanvas
- ✧ 直接在控件上绘图
- ✧ 将透明位图画到桌面上
- ✧ 截取屏幕图像
- ✧ 自定义字体



理解 Windows GDI 与 TCanvas



遇到难题

许多程序员觉得图形开发是编程中最令人愉快的事情。因为图形使得您的应用程序能够将自己画在屏幕上，同时恰当地应用图形功能，则会使您的应用程序更为突出。但是，如何将自己的艺术设计显示在您的计算机中呢？如何使用 Delphi 提供的强大的图形图像功能呢？Windows GDI 与 Delphi 中的 VCL Tcanvas 有什么联系呢？



钥匙在此

Windows 用一个“图形设备接口（GDI）”提供图形编程功能，而 Delphi 中的 TCanvas 则是该功能的再现，TCanvas 封装了 GDI，并使之更为易于使用。

1. 理解 GDI

GDI 是 Windows 标准图形编程的关键。像艺术家在画布上画画一样，需要某样东西在画布上绘制、涂抹和着色。界面就是我们要进行绘制的画布，可以通过 GDI 进入到这个窗口的界面，然后通过绘图方法以及绘图对象对这个画布进行绘制。记住，在这个窗口上绘图所需的任何东西都必须通过一个设备环境（Device Contex，即 DC，有时称之为设备上下文）进入其中。

GDI 函数组提供了在一个窗口界面上绘图的所有必须功能。这些功能使您在屏幕上能画出图形、文本和位图等。它们可以控制字体、颜色、线的粗细、阴影、比例、方向以及其他相关因素。

然而，绘制图形需要经过三步：首先必须获得要绘制窗口的设备环境，然后进行绘制，最后释放设备环境。注意最后一步比较重要，因为如果不释放该资源的话，至少会在应用程序中留下了一个资源漏洞，甚至会使您的系统进入“死亡蓝屏”的状态。

下面这个简单的例子，可以很简洁地说明如何使用 GDI 来绘制图形。该例子将坐标为（10,10）的像素变为红色。

```
var
    fDC:HDC;//声明一个设备环境句柄
begin
     fDC:=GetDC（fhandle）;//获得指定窗口的设备环境
```

```
    setPixel（fDC,10,10,RGB（255,0,0））;//在指定的设备环境中更改像素点的颜色
ReleaseDC（fDc）;//释放设备环境
End;
```

在这三个步骤中，第一步调用的是 GetDC（），该函数用于获得指定的窗口的设备环境句柄，其参数为指定窗口的句柄。第二步调用的是绘制图形的最基本的一个方法，在指定设备环境的位置上绘制一个像素点。最后一个函数是用于将所占用的设备环境释放掉。

2. Delphi VCL 使图形编程更为容易

Delphi 的 VCL 提供了 TCanvas 对象，它封装了 Windows GDI。TCanvas 处理所有的设备环境和幕后的释放资源的问题，因此大大简化了编程过程。例如，为了要达到上一部分同样的功能，以下一行代码即可以实现：

```
MyCanvas.Pixels（10,10）:=clRed。
```

如果确实需要使用设备环境的话，可以直接访问 TCanvas 的 Handle 属性。实际上，这就是该画布对象的句柄。例如，可以通过以下方法：

```
MyCanvas.Handle
```

此时，您也许已经感到了使用 Delphi 的 VCL 带来的方便，它使您将精力放到了所应该完成的任务上，而不是放到一些无谓的语句上去。

3. 使用 TCanvas

TCanvas 对象是像 Tform、TgraphicControl 及其后代这样类的一种标准属性。它提供一个接口，可以在此接口上进行绘图和重现对象以及控件。

TCanvas 对象用聚合来包含一定数量的与图形有关的类，这正如画家用来绘图的工具一样。可以通过以下属性访问它们：

- 画刷（TBrush） 这是一种用在多边形和椭圆形的内部的工具。用来在一个窗口或窗口化的控件的背景上涂色的图形工具，其类名为 TBrush。
- 字体（TFont） 字体是在屏幕上展示文本的方式。它控制着字体的高度、大小、名称和样式等，其类名为 Tfont。
- 画笔（Tpen）规定了用来画线及图形轮廓的画笔的种类，其类名为 Tpen。
- 像素 像素提供了对图像中某个位置像素点属性的访问，如像素颜色等。
- 限幅矩形（ClipRect）ClipRect 属性定义了用来在其中画图的特殊矩形。如果指定了一个限幅矩形的话，就不能在这个区域之外进行绘制，哪怕是这个图形比 TclipRect 大。
- CopyMode（复制模式） 这个属性决定了一个图像是如何从一个画布复制到另一个画布上去的。通过设置该属性可以改变将图形图像画到画布上的方式。

当然，TCanvas 不仅仅包含了这些绘图工具。



跟我来

通过上述的介绍，知道了通过 TCanvas 可以很容易地绘制出我们所需要的图形或图像。

上述的介绍才只是入了门，至于如何灵活使用 TCanvas，需要较大篇幅才可以介绍完的，限于篇幅，本章不对其进行详细的介绍，在 Delphi 帮助中通过输入 TCanvas 可以得到它完整的属性、方法和事件。对于它的使用可以参考本章所有的例子。

下面看一个例子，它可以在窗体上绘制出许多不同大小、样式和颜色的矩形。为了执行代码，可以将一个 Ttimer 组件放在窗体上，并使用对象观测器 object inspector 产生 OnTimer and OnActivate 事件。

清单 14-1　绘制矩形

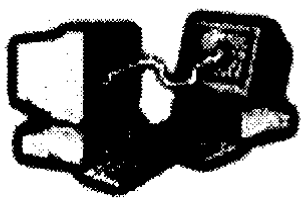
```
var

  X, Y: Integer;
procedure TForm1.FormActivate（Sender: TObject）;
begin
  WindowState := wsMaximized;
  Timer1.Interval := 50;
  Randomize;

end;

procedure TForm1.Timer1Timer（Sender: TObject）;

begin
  X := Random（Screen.Width - 10）;
  Y := Random（Screen.Height - 10）;
  Canvas.Pen.Color := Random（65535）;//为画笔产生颜色随机数
  case Random（5） of//设置画笔的随机类型
    0: Canvas.Pen.Style := psSolid;
    1: Canvas.Pen.Style := psDash;
    2: Canvas.Pen.Style := psDot;
    3: Canvas.Pen.Style := psDashDot;
    4: Canvas.Pen.Style := psDashDotDot;
  end;
  //根据上述定义的画笔来画出随机大小的矩形
  Canvas.Rectangle（X, Y, X + Random（400）, Y + Random（400））;
end;
```



直接在控件上绘图



遇到难题

标准的 Windows 控件对那些界面要求很高的多媒体软件来说还是逊色了不少。所以应用程序希望能够对这些控件在界面上有所改造，并可以按照自己的风格来实现。因此，如果能对这些控件进行界面上的重新绘图的话，就可以满足多媒体软件的一些要求了。



钥匙在此

在 Delphi 中为编程者提供了一个灵活的绘图场所,即前一节所述的 Canvas 类,在 Delphi 中很多控件都具有此属性,使编程者可以在这些控件的表面随心所欲地绘图,这对完善用户界面或者制作一些屏幕特技都有着非凡的作用。

要能够通过画布（Canvas）在这些控件上直接画图的话，首先需要这些控件带有画布属性。具有 Canvas 属性的常用控件有:

Tbitmap、TcomboBox、TDBComboBox、TDBGrid、TDBListBox、TdirectoryListBox、TdrawGrid、TfileListBox、Tform、TheaderControl、Timage、TlistBox、Toutline、TpaintBox、Tprinter、TstatusBar 以及 TStringGrid 等。

Tcanvas 类具有很多属性及命令，由于篇幅所限,省略了其他方法，以下方法的参数及格式的具体说明请参考文后程序及 Delphi 帮助文件。

canvas.rectangle（）:画矩形;
pen.color:定义画笔颜色;
roundrect（）:画圆角矩形;
arc（）:画弧线（不填充）;
chord（）:画弧线（填充）;
pie:画扇形;
polygon（）:画多边形（填充）;
polyline（）:多点连线（不填充）;
Pixels（）:指定固定像素点颜色值;
moveto（）:指明画线起点;
lineto（）:指明画线终点;
pen.width:定义画笔宽度;

brush.color:定义填充颜色；
textout（）:在固定位置输出字符串；
textwidth:取字符串高度；
textheight:取字符串宽度；
font.color:指定字体颜色；
font.size:指定字体大小；
Ellipse（）:画圆或椭圆。

使用 Canvas 的注意事项：当进行重画窗口时，画布上的图像将消失。比如当窗口进行最小化又重新恢复时，就会引起画布上图像的消失。另外当刚刚运行程序时，窗口也是属于重新绘制，所以如果在绘制窗口之前在画布上绘图，那么在弹出窗口后将不能显示出图像，比如在 TForm1.FormCreate（）事件中进行画布绘图操作将是徒劳的，但可以在此事件中进行画布操作的准备工作，比如设置画笔的颜色和宽度等。



跟我来

下面举例说明几种特殊屏幕效果的形成过程。注意，以下代码中可以省略掉 Form1 的引用，因为在该窗体中缺省的全部是引用的 Form1。

1. 为控件增加阴影或投影效果

基本原理就是利用该控件的父控件，比如 FORM 或者其他容器控件的 Canvas，在需要修饰的控件周围的适当处填加图像,通过线条及颜色的合理搭配，使控件与周围的图像融为一体,形成特殊的视觉效果,具体操作时需要根据被修饰控件的 TOP、LEFT、WIDTH 以及 HEIGHT 等属性，来确定需要画线的起点和终点坐标，这样操作无论被修饰控件的位置及尺寸如何变化，都可以保证投影及阴影效果完美地实现。比如可在窗口中建立三个按钮,然后在按钮 2 和按钮 3 的 Click 事件中添加如下代码，之后如果按下按钮 2 或按钮 3 便会使按钮 1 形成投影和阴影效果。

```
procedure TForm1.Button2Click（Sender: TObject）;
var
    x,y,i:integer;
begin
        x:=0;y:=0;
        form1.canvas.pen.width:=1;
        for i:=0 to 8 do
        begin
            form1.canvas.pen.color:=$00a0a0a0;
            form1.canvas.moveto（Button1.left+Button1.width+x,
                                Button1.top+y）;
            form1.canvas.lineto（Button1.left+Button1.width+x,
```

```
                Button1.top+button1.height+y）;
        form1.canvas.pen.color:=$00606060;
        form1.canvas.moveto（Button1.left+x,
                Button1.top+Button1.height+y）;
        form1.canvas.lineto（Button1.left+Button1.width+x,
                Button1.top+button1.height+y）;
        x:=x+1;
        y:=y+1;
    end;
end;

procedure TForm1.Button3Click（Sender: TObject）;
var
        x,y,i:integer;
begin
        x:=0;y:=0;
        form1.canvas.pen.width:=1;
        for i:=0 to 8 do
        begin
            form1.canvas.pen.color:=$00404040;
            form1.canvas.moveto（Button1.left+Button1.width+x,
                    Button1.top+8）;
            form1.canvas.lineto（Button1.left+Button1.width+x,
                    Button1.top+button1.height+8）;
            form1.canvas.moveto（Button1.left+8,
                    Button1.top+Button1.height+y）;
            form1.canvas.lineto（Button1.left+Button1.width+8,
                    Button1.top+button1.height+y）;
            x:=x+1;
            y:=y+1;
        end;

end;
```

2. 为控件加上边框

Delphi 中有很多控件没有边框属性，利用 Canvas 可以为任意的控件填加边框，从而使其轮廓清楚。具体操作时,可先定义出画笔的颜色和画笔的宽度（边框宽度），然后用 polyline（）命令根据相应控件的位置和尺寸自动定位四角坐标，一次可完成一个固定宽度矩形的绘制工作。如果需要多种颜色或者具有立体效果的边框，可多次定义画笔颜色，画出连续的多个矩形，通过调整相邻矩形的颜色来实现特殊效果的边框绘制工作。比如在窗体中安放两个按钮，并分别在 MouseMove 事件中填加如下代码,之后每当鼠标移到相应的按钮上时，相应

的按钮就会出现特殊颜色的边框，当鼠标移到窗口空白处时，则窗口会出现明显的边线，从而起到特殊的提示效果。

```
procedure TForm1.Button1MouseMove （Sender: TObject; Shift: TShiftState; X, Y: Integer）;
begin
form1.repaint;
form1.Canvas.pen.color:=clGreen;
form1.Canvas.pen.width:=5;
form1.Canvas.polyline（[point（Button1.left,Button1.top）,
point（Button1.left+Button1.width,Button1.top）,
point（Button1.left+Button1.width,Button1.top+Button1.height）,
    point（Button1.left,Button1.top+Button1.height）,
point（Button1.left,Button1.top）]）;
end;

procedure TForm1.Button2MouseMove （Sender: TObject; Shift: TShiftState; X,
  Y: Integer）;
begin
form1.repaint;
form1.Canvas.pen.color:=clBlue;
form1.Canvas.pen.width:=5;
form1.Canvas.polyline（[point（Button2.left,Button2.top）,
point（Button2.left+Button2.width,Button2.top）,
point（Button2.left+Button2.width,Button2.top+Button2.height）,
point（Button2.left,Button2.top+Button2.height）,
point（Button2.left,Button2.top）]）;
end;

procedure TForm1.FormMouseMove （Sender: TObject; Shift: TShiftState; X,
  Y: Integer）;
begin
form1.repaint;
form1.Canvas.pen.color:=clRed;
form1.Canvas.pen.width:=3;
form1.Canvas.polyline（[point（0,0）, point（form1.width-10,0）,
point（form1.width-10,form1.height-30）,
point（0,form1.height-30）, point（0,0）]）;
end;
```

3. 为图形按钮或其他控件安装动态帮助功能

此处主要使用字符串显示功能,也就是当鼠标移到指定控件上时,在其周围的适当位置显示相应的字符串,当鼠标离开时通过图像覆盖动作使字符串消失。这可以使用一个简单的命

令即 form1.repaint 完成，它的作用是重新绘制 FORM1 内的所有控件，这样便可快速清除原来显示的字符。编程时主要使用各个控件的 MouseMove（）事件。比如可在窗口中安放两个按钮,并安排如下代码,运行之后当鼠标移到相应按钮上时,就会在其上方显示相应的提示信息，当鼠标移到窗口空白处则会在窗口的左下角显示"OK"字样。

```
procedure TForm1.Button1MouseMove（Sender:TObject;Shift:TShiftState;X, Y: Integer）;
begin
    form1.repaint;
    form1.Canvas.brush.color:=clGreen;
    form1.Canvas.font.size:=8;
    form1.Canvas.font.color:=$1200ffff;
    form1.Canvas.TextOut（Button1.left,
        Button1.top-15,'OPEN FILE'）;
end;

procedure TForm1.Button2MouseMove （Sender:TObject;Shift:TShiftState;X, Y: Integer）;
begin
    form1.repaint;
    form1.Canvas.font.size:=8;
    form1.Canvas.brush.color:=clRed;
    form1.Canvas.font.color:=$1200ffff;
    form1.Canvas.TextOut（Button2.left,Button2.top-15,'EXIT'）;
end;

procedure TForm1.FormMouseMove （Sender:TObject;Shift:TShiftState;X,
  Y: Integer）;
begin
    form1.repaint;
    form1.Canvas.font.size:=12;
    form1.Canvas.brush.color:=clBlue;
    form1.Canvas.font.color:=$00ffffff;
    form1.Canvas.TextOut（0,form1.height-50,'OK'）;
end;
```

4. 模仿电视移动字幕或图像

用此方法制作移动字幕的基本原理是：首先选定一个可作为画布的控件,比如 FORM 或 IMAGE,然后用一种填充颜色填满整个画布，并在画布的固定位置显示一个特殊前景色的固定尺寸的字符串，接着在 TIMER 控件中定时连续改变显示字符串的位置。由于原来位置的字符串不能自动消失,所以需要用与填充颜色相同颜色的线条覆盖多余部分，这样便可实现平滑的字幕移动效果。实现图像移动的方法:可事先定义 Bitmap 类和 Rect 类变量，然后在程序运行阶段建立 Bitmap 和 Rect 对象,并在 Bitmap 中装入指定的图像文件,同时把对象 Bitmap 装入 Rect 对象中以显示图像，然后在 Timer 控件中定时连续改变 Rect 的坐标值，并

用与填充颜色相同的线条覆盖遗留图像部分，即可以实现图像的平稳移动。采用上述方法实现的字符串或图像移动操作，其效果平稳，无任何闪烁或抖动现象，视觉效果极佳，并且不需要事先建立控件，完全由程序代码实现，具有相当多的优点。

编程举例:在窗体中安放时间控件 Timer1 和图像框 Image1，定义 TRect 及 TBitmap 型变量,并在 FormCreate 和 Timer1Timer 事件中安排如下代码,程序运行之后即会在图像框中出现自下而上的移动字幕及图像效果,并从顶部消失,之后重新从底部出现。

```
var    { 定 义 变 量}
  Form1: TForm1;
  x,tt,l,h:integer;
  pic:Trect;
  map:Tbitmap;
{ 具 体 动 画 演 示 代 码}
procedure TForm1.Timer1Timer（Sender: TObject）;
begin
x:=x-1;
tt:=tt-1;
if x<-160 then
x:=image1.height+20;
tt:=x+80;
image1.Canvas.font.size:=18;
image1.Canvas.font.color:=$1200ffff;
image1.Canvas.TextOut（10,x,'OK 字 幕 移 动 演 示'）;
image1.Canvas.font.size:=12;
image1.Canvas.font.color:=$120000ff;
image1.Canvas.TextOut（20,x+50,' 字 体 变 色 效 果 演 示'）;
l:=image1.Canvas.textwidth（' 字 体 变 色 效 果 演 示'）;
h:=image1.Canvas.textheight（' 字 体 变 色 效 果 演 示'）;
image1.Canvas.pen.color:=clGreen;
image1.Canvas.moveto（20,h+x+50-2）;
image1.Canvas.lineto（20+l,h+x+50-2）;
pic.topleft.x:=30;
pic.topleft.y:=tt;
pic.bottomright.x:=pic.topleft.x+100;
    pic.bottomright.y:=pic.topleft.y+80;
image1.canvas.stretchdraw（pic,map）;
image1.Canvas.pen.color:=clGreen;
image1.Canvas.moveto（0,pic.topleft.y+80）;
image1.Canvas.lineto（pic.topleft.x+100,pic.topleft.y+80）;
end;
```

```
{以下设置动画初值}
procedure TForm1.FormCreate（Sender: TObject）;
begin
timer1.enabled:=true;
timer1.Interval:=10;
x:=image1.height+20;
tt:=x+80;
form1.repaint;
image1.Canvas.brush.color:=clGreen;
pic:=Rect（0,0,image1.width,image1.height）;
image1.Canvas.FillRect（pic）;
pic:=Rect（-1,-1,1,1）;
map:=Tbitmap.create;
map.loadfromfile（'d:\windows\256color.bmp'）;
image1.canvas.stretchdraw（pic,map）;
end;
```



将透明位图画到桌面上



遇到难题

Windows 桌面对于程序员来说，就是一个画布，在其上进行画图的话，将是一件比较愉快的事情，可以将其画得花里胡哨（当然，这需要您的艺术天分了），也可以将其画得乱七八糟。在 Office 2000 中，我们可以看到一个小助手（小动画图形）在桌面上晃来晃去。国外还有一些小软件，也具有类似的功能。例如，经常看到“豆子”先生在屏幕上直接走来走去的。这是怎么实现的呢？当然，要实现这些功能还需要动画的一些知识，为了起到抛砖引玉的功效，本节只讲述如何将一个透明位图画到桌面上。



钥匙在此

为了将透明位图画到桌面上，需要两个技术，一是如何实现位图透明，二是如何将所获得的位图画到桌面上。下面分别描述。

1. 实现位图的透明化

用过 Timage 的程序员也许已经知道，在 Timage 对象中有一个 Transparent 属性，可以

实现位图透明化。当将该属性设为 True 时，即可实现位图的透明化。然而，事情往往不可两全其美，该属性使用起来虽然非常简单，但是，要实现更深层次应用时就无能为力了。如果使用这个属性实现位图的透明化的话，程序员不免要问了，要透明它是如何确定透明色的呢？经过笔者的试验发现，Timage 是以位图的左上角（0,0）的颜色为透明色而将整个位图透明化的。但有时这种结局往往不是我们所希望的，透明色可以指定才是我们设计的目标。

为了实现透明位图，需要经过一些逻辑运算。对于编程来讲所需要的参数应该是三个部分，即源位图、透明色和目标位图。实现透明色的原理是：先将源位图与透明色进行逻辑与运算，再进行反色处理，然后将结果与源位图进行与操作，即获得透明色与源位图的异或处理位图，再进行一些逻辑运算最终获得源位图与目标位图合二为一的位图，并使源位图看上去像贴在了目标位图上一样，并且将指定的颜色变为透明色，该过程用代码描述如下。

```
{该过程将源位图绘制到目标画布上
 thandle    = 目标画布句柄
 x,y = 所要画的目标位置
 s    = 源位图
 TrCol = 在源位图上变成透明的颜色，也就是使什么样的颜色变成透明
 NOTE: 不要重新刷新目标位图的设备}

procedure DrawTransparent（thandle: HDC; x,y: Integer; s: TBitmap; TrCol: TColor）;
var
  bmpXOR, bmpAND, bmpINVAND, bmpTarget: TBitmap;
  oldcol: Longint;
begin
  try
    bmpAND := TBitmap.Create;
    bmpAND.Width := s.Width;
    bmpAND.Height := s.Height;
    bmpAND.Monochrome := True;
    oldcol := SetBkColor（s.Canvas.Handle, ColorToRGB（TrCol））;
    //将源位图与透明色进行与操作并存入 bmpAND
   BitBlt（bmpAND.Canvas.Handle, 0,0,s.Width,s.Height, s.Canvas.Handle, 0,0,
          SRCCOPY）;
    SetBkColor（s.Canvas.Handle, oldcol）;

    bmpINVAND := TBitmap.Create;
    bmpINVAND.Width := s.Width;
    bmpINVAND.Height := s.Height;
    bmpINVAND.Monochrome := True;
    //将源位图进行反向拷贝
    BitBlt（bmpINVAND.Canvas.Handle, 0,0,s.Width,s.Height,
  bmpAND.Canvas.Handle, 0,0, NOTSRCCOPY）;
```

```
    bmpXOR := TBitmap.Create;
    bmpXOR.Width := s.Width;
    bmpXOR.Height := s.Height;
    BitBlt（bmpXOR.Canvas.Handle, 0,0,s.Width,s.Height, s.Canvas.Handle, 0,0,
SRCCOPY）;
    //获取异或位图
    BitBlt（bmpXOR.Canvas.Handle, 0,0,s.Width,s.Height,
bmpINVAND.Canvas.Handle, 0,0, SRCAND）;

    bmpTarget := TBitmap.Create;
    bmpTarget.Width := s.Width;
    bmpTarget.Height := s.Height;
    //将目标位图拷贝到 bmpTarget 中
    BitBlt（bmpTarget.Canvas.Handle, 0,0,s.Width,s.Height, tHandle, x,y,
SRCCOPY）;
    //将经与源位图异或的位图同 bmpTarget 进行与操作
    BitBlt（bmpTarget.Canvas.Handle, 0,0,s.Width,s.Height,
bmpAND.Canvas.Handle, 0,0, SRCAND）;
    //进行反向操作
    BitBlt（bmpTarget.Canvas.Handle, 0,0,s.Width,s.Height,
bmpXOR.Canvas.Handle, 0,0, SRCINVERT）;
    //拷贝至目标设备上下文中
    BitBlt（tHandle, x,y,s.Width,s.Height, bmpTarget.Canvas.Handle, 0,0,
SRCCOPY）;
  finally
    bmpXOR.Free;
    bmpAND.Free;
    bmpINVAND.Free;
    bmpTarget.Free;
  end;{End of TRY section}
end;
```

对于其中使用的函数解释如下：

（1）SetBkColor（）设置背景色。该函数为一个 GDI 函数，其格式为：

```
COLORREF SetBkColor（
    HDC hdc,// 设备上下文句柄
    COLORREF crColor    // 背景颜色值
  ）;
```

如果设置成功则返回原背景色，否则，返回 CLR_INVALID 值。

（2）BitBlt（）函数。将像素矩形中的颜色数据从指定的源设备上下的文执行位块

（bit-block）传输到目标设备上下文中。其定义格式为：

```
BOOL BitBlt（
    HDC hdcDest,//目标设备上下文句柄
    int nXDest,    //目标矩形的左上角 x 坐标
    int nYDest,    //目标矩形的左上角 y 坐标
    int nWidth,    //目标矩形的宽度
    int nHeight,   //目标矩形的高度
    HDC hdcSrc,  // 源设备上下文句柄
    int nXSrc,     // 源矩形的左上角 x 坐标
    int nYSrc,     // 源矩形的左上角 y 坐标
    DWORD dwRop    // 光栅操作代码
  ）;
```

其中光栅操作代码定义了源矩形中的颜色数据如何与目标矩形中的颜色数据进行组合的，并得到最终的颜色。表 14-1 显示了一些常用的光栅操作代码。

表 14-1　　常用的光栅操作代码

值	描　述
BLACKNESS	用黑色进行填充
NOTSRCCOPY	进行反向拷贝
SRCAND	进行与操作
SRCCOPY	进行直接的拷贝
SRCPAINT	进行或操作
SRCINVER	进行异或操作

（3）ColorToRGB（）函数。该函数用于将颜色值转化为 RGB 的显示方式。其声明格式为：

```
function ColorToRGB（Color: TColor）: Longint;
```

2. 获取桌面设备环境（设备上下文）句柄

为了获取桌面设备环境的句柄，可以调用 GDI 中 GetDC（）函数来实现。该函数可以为指定的窗体客户端区域获得显示设备环境的句柄，该值可以被其他的 GDI 函数所使用，如 1.（2）中的 BitBlt（）函数。GetDC（）的声明格式为：

```
HDC GetDC（
    HWND hWnd // 窗口句柄
  ）;
```

返回值为指定窗口的设备环境句柄。如果要获取 Windows Desktop 桌面的设备环境句柄的话，直接指定参数为 0 即可，如 GetDC（0）返回的就是 Windows 桌面的设备环境句柄。



跟我来

现在使用上述定义的函数来实现在桌面上绘图。首先新建一个应用程序，并在其窗体

上放置一个按钮和 Timage 控件，然后在按钮单击事件中添加使用上述函数，为了做一个对比，在窗体上也画出透明位图，其代码如下：

```
procedure TForm1.Button1Click（Sender: TObject）;
begin
    //在桌面上绘制位图。GetDC（0）获取桌面的设备环境
    DrawTransparent（GetDC（0）, 0,0,image1.Picture.Bitmap, clblack）;
end;

procedure TForm1.FormPaint（Sender: TObject）;
begin
    //在窗体上绘制位图
    DrawTransparent（canvas.handle, image1.Width,0,image1.Picture.Bitmap, clblack）;
end;
```

运行该程序，并单击按钮时程序会将透明位图画在桌面上，由于窗体的刷新事件，会将位图画在窗体上，结果如图 14-1 所示。

图 14-1 将透明图画在桌面上以及窗体上。

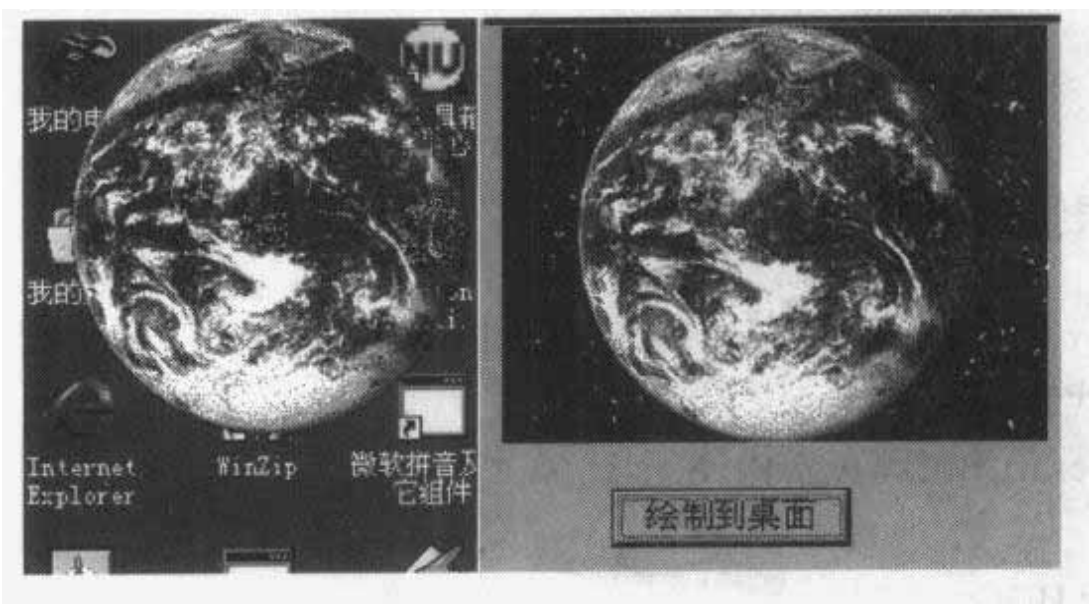


图 14-1 透明图画在桌面上



截取屏幕图像



遇到难题

在第三节中实现了将位图画到桌面上，有时想将该过程倒过来，也就是将桌面图像截取到图程序中。这也是完全可以实现的。



钥匙在此

为了截取桌面图像，同样需要三步，首先需要获得桌面画布的句柄，然后将桌面图像复制到存放图像的组件中，最后释放所占的资源。

为了获得桌面画布的句柄，可以使用 GDI 中的 CreateDC（）函数。通过指定设备的名称来创建设备环境（上下文）。其声明格式如下：

```
HDC CreateDC（
    LPCTSTR lpszDriver,  // pointer to string specifying driver name
    LPCTSTR lpszDevice, // pointer to string specifying device name
    LPCTSTR lpszOutput, // do not use; set to NULL
    CONST DEVMODE *lpInitData      // pointer to optional printer data
）;
```

对于显示设备来说，只需指定第一个参数，并将之设成“DISPLAY”，其余参数都设成空即可。返回值为显示设备的句柄。

例如，ScreenDC:=CreateDC（'DISPLAY',Nil,Nil,Nil）；

要实现位图的复制同样要使用 BitBlt（）函数，并将光栅操作参数指定为 SRCCOPY。

当调用了该函数之后，最后不要忘了将该设备占用的资源删除掉。使用 DeleteDC（）便可以将占用的资源删除掉。



跟我来

创建一个新应用程序，然后在窗体上放置 Timage 组件以及一个按钮，并对按钮的单击事件进行编程，代码如下：

```
procedure TForm1.BitBtn1Click（Sender: TObject）;
Var
  ScreenDC : HDC;
Begin
  //获取桌面设备环境
  ScreenDC:=CreateDC（'DISPLAY',Nil,Nil,Nil）;
    //将桌面图像复制到 Timage 的画布中
BitBlt（Image1.Canvas.Handle, 0,0, Image1.Width, Image1.Height,
     ScreenDC, 0,0, SRCCOPY）;
  Image1.Refresh;
    //释放所占资源
  DeleteDC（ScreenDC）;
end;
```

运行该程序，单击“截取屏幕”按钮便可以将桌面图像截取并放置到 Timage 组件中，其结果如图 14-2 所示。



图 14-2　截取桌面图像



再送礼包

为了获得桌面设备环境，还有其他方式。例如，如果能获得桌面窗口句柄的话就可以通过 GetDC()获取桌面设备环境，而获得桌面窗口句柄可以通过 GDI 函数 GetDesktopWindow 来实现。

```
ScreenDC:=GetDC（GetDesktopWindow）;
```

该语句与 ScreenDC:=CreateDC（'DISPLAY',Nil,Nil,Nil）语句作用相同。



自定义字体



遇到难题

字体在界面设计中同样起到非常重要的作用，但是，Windows 中预提供的字体有时会满足不了要求。例如，有时需要将字体进行一定角度的旋转，或者需要一个特殊形状的字体等等，此时，就需要进行编程来实现艺术构思了。



钥匙在此

要创建新字体，需要使用到 GDI 的 CreateFontIndirect（）函数和 TLogFont 数据结构。创建新字体，需要两个步骤，先填写新字体的逻辑结构，然后根据所定义的字体逻辑

结构来创建新字体。现在阐述如下。

1. 填写新字体的逻辑结构 TlogFont

TlogFont 是一个记录结构类型，用于描述字体的信息。为了创建新字体，首先要做的是填写新字体的逻辑结构，因此，程序员首先应该掌握有关 TlogFont 结构的知识。这个结构在 Windows.pas 单元中是这样进行声明的：

```
TlogFont= packed record
  lfHeight: Longint;
  lfWidth: Longint;
  lfEscapement: Longint;
  lfOrientation: Longint;
  lfWeight: Longint;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..LF_FACESIZE - 1] of AnsiChar;
end;
```

TlogFont 结构的每个域都是用于指定字体的有关属性的。每个域代表一种字体的属性。缺省情况下，大多数域设为 0。一般的规律是，填写的域越多，就可以更细致地控制字体的样式。下面这个表说明了 TlogFont 结构中每个域的含义。关于这些域的进一步说明，请参见 Win32 的帮助。这里只介绍几个最常用的域。

表 14-2　　TlogFont 结构说明

字段值	描 述
lfHeight	字体高度。大于零的值表示单元的高度，小于零的值表示图形高度。设为零，表示让 Win32 决定高度
lfWidth	字体的平均宽度。设为零，表示让 Win32 决定字体的宽度
lfEscapement	字体基准线的旋转度（以 10 度为单位）。在 Windows 95/98 中，文本和单个字符使用相同的角度输出。这就是说，lfEscapement 和 lfOrientation 相等。在 Windows NT 中，字符串中的文本以不同的角度输出。对于后者说，必须调用 Win32 GDI 函数 SetGraphicsMmode（）把它设为 GM_ADVANCED。缺省情况下，图形模式为 GM_COMPATIBLE，它使得 Windows 95/98 与 WindowsNT 行为一样。这种字体旋转效果只对 TrueType 字体有效
lfOrientation	指定输出单个字符的角度。在 Windows 95 中，其值与 lfEscapement 相等。在 WindowsNT 中，其值可能不同
lfWeight	这个域值确定字体的密度。Windows 单元中为这个域定义了若干个变量，诸如 FW_BLOD 和 FW_NORMAL。如果设为 FW_DONTCARE，表示让 Win32 选择磅值

续表

字段值	描 述
lfItalic	非零表示斜体，零表示非斜体
lfUnderline	非零表示有下划线，零表示没有下划线
lfStrikcOut	非零表示有一线穿过字体，零表示没有
lfCharSet	Win32 定义了字符集：ANSI_CHARSET=0,DEFAULT_CHARSET=1; SYMBOL_CHARSET=2,SHIFTIS_CHARSET=128,OEM_CHARSET=255
LfOutPrecision	指定 Win32 如何使要创建的字体的大小和属性与实际的字体相匹配
LfClipPrecision	指定 Win32 如何去裁减字符
Lfquality	指定输出字符的质量
lfpitchAndFamily	以低两位定义字体的斜度，在高四位指定字体的家族
lfFaceName	字体的样式名

2. 创建新字体的函数 CreateFontIndirect（）

当上述的逻辑字体信息填写完后，可以调用创建字体函数对新定义的字体进行创建。CreateFontIndirect（）函数的声明格式为：

function CreateFontIndirect（const p1: TLogFont）: HFONT;

其参数为逻辑字体类型的常量。返回值为新创建字体的句柄。



跟我来

下面例子是通过对当前字体进行一些修改来创建新字体的。

若想以当前字体作为起点，可使用 Windows 的 GetObject()函数检查当前字体的 TlogFont 结构，然后通过对该结构进行适当的修改来创建新的字体。仅简单地在一个 VCL 构件上替换字体的句柄并不能保证正常工作，更好的办法是创建一个并不与当前控件相关的新 Tfont，并做一些改变，然后把字体赋给该控件。

这里是一个创建旋转 45 度的新字体例子。

```
procedure TForm1.Button1Click（Sender: TObject）;
var
    lf:TLogFont;
    tf:TFont;
begin
    tf:=TFont.Create;
    //将当前的字体赋给新字体
    tf.Assign（button1.font）;
    //获取所取字体的逻辑字体结构
    GetObject（tf.handle,sizeof（lf）,@lf）;
    //改变逻辑字体结构的间隔和方向成员的值，使字体旋转 45 度
    lf.lfEscapement:=450;
    lf.lfOrientation:=450;
```

```
    //根据逻辑字体创建新字体
    tf.handle:=CreateFontIndirect（lf）;
    Button1.ParentFont:=false;
    //将新字体赋给按钮
    Button1.Font.Assign（tf）;
    tf.Free;
end;
```

该例子创建了一个新的 Tfont，并把按钮的当前字体赋给此 Tfont，这将把按钮字体的所有属性都复制到这个新的 Tfont 中，然后调用 Windows 的 GetObject（）函数来检索该 Tfont 的逻辑字体结构，接着改变该逻辑字体结构的间隔和方向成员以使字体旋转 45 度，这可以通过 1/10 度为单位来说明角度。

然后利用 Windows 的 GDI 命令 CreateFontIndirect（）根据逻辑字体结构创建一个新字体。该命令的返回值是我们以前赋给 Tfont 的 windows 字体句柄。最后把按钮的 ParentFont 置为 False，以使按钮不再继承表格的字体属性，并把新建的 Tfont 赋值给按钮的字体，且释放 Tfont。应注意的是选择旋转的字体须能旋转才可以，因为并非所有的字体都可旋转。建议使用 TrueType 字体，因为该字体总是可旋转的。本例中选择的是“宋体”。



不是所有的字体类型都能旋转的，如果字体前标志为“T”（TrueType 字体）的，则可以旋转。

运行该代码，并单击按钮可以获得以下结果，如图 14-3 所示。



图 14-3　旋转字体

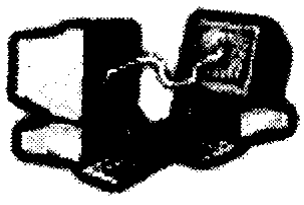
第十五章

多媒体编程

热点透视

在多媒体的应用中，许多文章只是一般地介绍TmediaPlayer控件是如何使用的，而TmediaPlayer则是MCI的一个封装，但对于其他低层的控制的介绍文章并不多，本章则力求实用并具有一定深度地进行多媒体编程，介绍光驱的控制和播放、视频信息的捕获以及有关屏幕保护程序方面的一些编程。内容主要包括：

- 控制光驱
- 设置光驱自动播放功能
- 获取Audio CD信息
- 捕获视频图像
- 制作屏幕保护程序
- 打开和关闭屏幕保护程序



控制光驱



遇到难题

在很多媒体播放器中有光驱弹出和关闭等控制按钮，例如，Xing、Power DVD 等 VCD 或 DVD 播放器中就有这些按钮。有时在自己的程序中也同样需要控制光驱。例如，自己有一个多媒体教学盘，当该盘播放结束后需要更换另一张光盘时应当自动弹出光驱托盘；当用户将盘放入其中时可以让软件来控制光驱的关闭，而不是靠硬件按钮来操作光驱。因此，整个过程都是可以通过软件来实现对光驱的控制。



钥匙在此

对光驱的控制，一般包含两个操作，一是打开光驱，另一个是关闭光驱。

要实现对光驱的控制，需要经过两个步骤，首先应该获取光驱设备的 ID 号，其次，向该设备发送打开或者关闭媒体的命令消息。下面对这两个步骤进行详细阐述。

1. 获取光驱设备号

Delphi 的 TmediaPlayer 元件的确是个好东西，它非常精巧。在这个小巧的控件中，封装了 Windows 媒体控件接口（MCI）的大量函数。MCI 是 Windows API 的一部分，用于控制多媒体设备。

TmediaPlayer 支持大量被 MCI 支持的媒体设备。TmediaPlayer 支持的设备类型可以通过 DeviceType 特性来确定。表 15-1 列出了 DeviceType 特性的可能值。

表 15-1　　DeviceType 特性

设备类型	描　述
DtAutoSelect	TmediaPlayer 根据要播放的文件名自动选择适当的设备类型
DtAVIVideo	AVI 文件。这些文件的扩展名是.avi，含有声音和视频
dtCDAudio	在计算机的 CD-ROM 驱动器中播放的音频 CD
DtDat	与 PC 机相连的数字录音机（DAT）
DtDigitalVidio	数字视频设备，如数字相机
DtMMMovie	多媒体电影格式
DtOther	未确定的多媒体格式
DtOverlay	视频特技设备
DtScanner	与 PC 相连的扫描仪
DtSequencer	能播放 MIDI 文件的设备。MIDI 文件的扩展名是 MIDI 或 RMI
DtVCR	与 PC 机相连的录像机（VCR）

续表

设备类型	描　述
DtVideodisc	与 PC 机相连的视频播放机
dtWaveAudio	Wave 音频文件

为了获得光驱设备号，首先应该指定其设备类型 DeviceType 为 dtCDAudio，然后调用 TmediaPlayer.Open 打开设备，最后可以通过只读属性 TMediaPlayer.DeviceID 来获得媒体设备号。

设当前的 TmediaPlayer 实例为 mp，上述获取光驱设备号的整个过程可以由代码描述如下：

```
mp.DeviceType := dtCDAudio;
  //指定文件所在地
mp.FileName := Drive + ':';
  //打开设备，获得设备 ID 号
mp.Open;
```

此时可以通过 mp.DeviceID 获取光驱的设备号。

2. 发送打开或关闭媒体的命令消息

当得到光驱的设备号之后，可以通过 Win32 API 函数 mciSendCommand（）向该设备发送设置光驱状态的命令来控制光驱。mciSendCommand（）可以将命令消息发送到指定的 MCI 设备，它在 MMSystem.pas 中的声明格式如下：

```
function mciSendCommand（mciID: MCIDEVICEID; uMessage: UINT;
  dwParam1, dwParam2: DWORD）: MCIERROR;
```

其中参数 mciID 表示将要获得命令消息的 MCI 设备的 ID 号，uMessage 为命令消息标志，剩下两个参数为命令消息的参数值。其中，Umessage 应该指定为 MCI_SET 命令来设置设备信息，该命令包括了对 CD audio、digital-video、MIDI sequencer、VCR、videodisc、video-overlay 以及 waveform-audio 等设备的设置。函数返回值为 0 时表示发送成功。

当命令消息为 MCI_SET 时，MCI 的发送函数可以表示为：

```
MCIERROR mciSendCommand（MCIDEVICEID wDeviceID, MCI_SET,
  DWORD dwFlags, （DWORD） （LPMCI_SET_PARMS） lpSet）;
```

参数 dwFlags 值，可以取表 15-2 所列的值（只列举了一部分，其余可以参考 Win32 SDK 帮助文档，也可以查 MCI_SET 命令消息），这些值对所有的设备都支持。

表 15-2　　MCI_SET 指令中参数 dwFlags 值

值	描　述
MCI_SET_DOOR_CLOSED	关闭媒体盖子，对于光驱而言为光驱托盘
MCI_SET_DOOR_OPEN	打开媒体盖子，对于光驱而言为光驱托盘
MCI_SET_OFF	使指定的视频和声频信道无效
MCI_SET_ON	使指定的视频和声频信道有效



跟我来

新建一个应用程序，然后在窗体上放置两个按钮，一个用于打开光驱，另一个用于关闭光驱，并在两个按钮中写入相应的代码。完整程序见清单 15-1。

清单 15-1　控制光驱

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, MPlayer;

type
  TForm1 = class（TForm）
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click（Sender: TObject）;
    procedure Button2Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
uses MMSystem;

function IsDriveCD（Drive : char）  : longbool;
var
  DrivePath : string;
```

```
begin
  DrivePath := Drive + ':\';
  //测试是否为光驱
  result := LongBool（GetDriveType（PChar（DrivePath））  and DRIVE_CDROM）;
end;

//弹出光驱
function EjectCD（Drive : char）  : bool;
var
  mp : TMediaPlayer;
begin
  result := false;
  Application.ProcessMessages;
  if not IsDriveCD（Drive）  then exit;
  mp := TMediaPlayer.Create（Nil）;
  mp.Visible := false;
  mp.Parent := Application.MainForm;
  //为了使其他控件或者应用程序也能使用指定的设备，因此将共享设为 True
  mp.Shareable := true;
  //指定设备类型为 dtCDAudio
  mp.DeviceType := dtCDAudio;
  //指定文件所在地
  mp.FileName := Drive + ':';
  //打开设备，并获得设备 ID 号
  mp.Open;
  Application.ProcessMessages;
  //向 CDAudio 设备发送打开媒体的命令
  mciSendCommand（mp.DeviceID,
  MCI_SET, MCI_SET_DOOR_OPEN, 0）; //该代码功能与 mp.Eject 一样
  Application.ProcessMessages;
  mp.Close;
  Application.ProcessMessages;
  mp.free;
  result := true;
end;

//关闭光驱
function CloseCD（Drive : char）:boolean;
var
  mp : TMediaPlayer;
```

```
begin
   result := false;
   Application.ProcessMessages;
   mp := TMediaPlayer.Create（Nil）;
   mp.Visible := false;
   mp.Parent := Application.MainForm;
   mp.Shareable := true;
   mp.DeviceType := dtCDAudio;
   mp.FileName := Drive + ':';
   mp.Open;
   Application.ProcessMessages;
   //向 CdAudio 设备发送关闭媒体的命令
   mciSendCommand（mp.DeviceID,
   MCI_SET, MCI_SET_DOOR_CLOSED, 0）;
   Application.ProcessMessages;
   mp.Close;
   Application.ProcessMessages;
   mp.free;
   result := true;
end;

procedure TForm1.Button1Click（Sender: TObject）;
begin
   if not EjectCD（'F'） then
      ShowMessage（'Not A CD Drive'）;
end;

procedure TForm1.Button2Click（Sender: TObject）;
begin
      CloseCD（'F'）;
end;

end.
```



再送礼包

在解决上述问题之后，还需要作一些补充说明，比如简要介绍解决该问题的其他方法，或者介绍本题的关键技术的扩展应用等。



总结

以上是解决每一个问题所应该包括的 5 个部分，各部分需要包含的内容见相应的说明，其中“再送礼包”和“总结”部分可以没有。



设置光驱自动播放功能



遇到难题

在 Windows 95/98 中，当光驱设置为自动播放状态时，如果插入的光盘为 AudioCD 或者在光盘中带有 Autorun.inf 文件，则该光盘可以自动启动相关的执行程序。当然，Windows 提供了是否自动播放的选项，但有时在应用程序中需要读取该选项或者需要对其通过程序代码进行设置。如何达到这一目标呢？



钥匙在此

对于光驱是否自动播放，其设置选项存放在了注册表的

Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\NoDriveTypeAutoRun

键中。该键的值为二进制数，总共四个字节，你可以通过检查这四个字节的值来确定光驱是否自动播放。例如，我目前的机器值为 df 00 00 00。

设置该键的下列位值可能会引起 Windows 不使用任何自动运行的能力。

DRIVE_UNKNOWN = bit # 0

DRIVE_NO_ROOT_DIR = bit #1

DRIVE_REMOVABLE= Bit # 2

DRIVE_FIXED = Bit #3

DRIVE_REMOTE= Bit #4

DRIVE_CDROM= Bit #5

DRIVE_RAMDISK= Bit #6

您可以通过测试一个你正在软盘或者网络驱动器上开发的应用程序自动运行的能力来设置这个值，如设置从软盘（DRIVE_REMOVEABLE）或者网络驱动器（DRIVE_REMOTE）

启动自动执行程序。如果该设备没有自动插入检查能力的话，只需简单的插入该盘然后在 Windows 浏览器中按【F5】即可引起自动执行程序的启动。



跟我来

下面的例子演示了自动检测 CD 是否有效的功能，并且也演示了打开自动检测和关闭自动检测 CD 的功能。

新建一个应用程序，并放置三个按钮在窗体上，然后输入如下代码，见清单 15-2。

清单 15-2　设置光驱自动执行选项

```
uses
  Registry;

function IsCdAutoRunOn : bool;
var
  reg: TRegistry;
  AutoRunSetting : integer;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_CURRENT_USER;
  reg.OpenKey（
    'Software\Microsoft\Windows\CurrentVersion\Policies\Explorer',
                false）;
    //读取注册表中二进制的值
  reg.ReadBinaryData（'NoDriveTypeAutoRun',
                      AutoRunSetting ,
                      sizeof（AutoRunSetting））;
  reg.CloseKey;
  reg.free;
  //检测 DRIVE_CDROM 是否被设为 0
  result :=  not （（AutoRunSetting and （1 shl 5））  <> 0）;
end;

procedure SetCdAutoRun（bOn : bool）;
var
  reg: TRegistry;
  AutoRunSetting : integer;
begin
  reg := TRegistry.Create;
```

```
reg.RootKey := HKEY_CURRENT_USER;
reg.LazyWrite := false;
reg.OpenKey（
  'Software\Microsoft\Windows\CurrentVersion\Policies\Explorer',
              false）;
reg.ReadBinaryData（'NoDriveTypeAutoRun',
                    AutoRunSetting ,
                    sizeof（AutoRunSetting））;
//根据本过程的参数设置光驱自动执行选项
if bOn then
  AutoRunSetting := AutoRunSetting and not （1 shl 5）
else
  AutoRunSetting := AutoRunSetting or （1 shl 5）;
  reg.WriteBinaryData（'NoDriveTypeAutoRun',
                       AutoRunSetting ,
                       sizeof（AutoRunSetting））;
  reg.CloseKey;
  reg.free;
end;

procedure TForm1.Button1Click（Sender: TObject）;
begin
  if IsCdAutoRunOn then
    ShowMessage（'CD AutoRun Is On'） else
    ShowMessage（'CD AutoRun Is Off'）;
end;

//打开自动执行选项
procedure TForm1.Button2Click（Sender: TObject）;
begin
  SetCdAutoRun（True）;
end;

//关闭自动执行选项
procedure TForm1.Button3Click（Sender: TObject）;
begin
  SetCdAutoRun（False）;
end;
```



获取 Audio CD 信息



遇到难题

在多媒体程序中能够播放 Audio CD 是应该具有的一个最基本的功能，那又如何使应用程序更加智能化呢？例如，检测所制定的光驱中是否包含 Audio CD、控制 Audio CD 的自动执行以及获取 Audio CD 的序列号信息等等，通过这些智能就会使得应用程序更加完整，从而使用起来也就会更加方便。



钥匙在此

下面分项进行阐述。

1. 怎样检测所给定的 CD-ROM 中是否带有 Audio CD

首先调用 WinAPI 函数 GetDriveType（）来判断所给定的驱动器是否为 CD-ROM 驱动器。然后，通过 API 函数 GetVolumeInformation（）检测其卷标名是否包含有“Audio CD”，如果有的话，则说明该 CD 为音乐 CD。

2. 如何查询 Win 32 中 Audio CD 的自动执行特征及如何设置它

这个设置值是存储在下列两个 Windows 注册表的键值中。

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AudioCD\Shell

HKEY_CLASSES_ROOT\AudioCD\Shell

如果自动执行的特征有效的话，则该值将是字符串“play”。如果要设置它的话，只要对注册表相应的键值进行设置，然后写回注册表即可实现。

3. 如何获取 Audio CD 的序列号

在 CD 中会嵌入有一个序列号或者国际产品代号。可以通过 Windows 的 MCI 指令来获取有关 Audio CD 的信息。该指令为 MCI_INFO_MEDIA_IDENTITY。



跟我来

根据上述的方法和技术，下面举几个相应的例子。

1. 检测所给定的 CD-ROM 中是否带有 Aúdio CD

新建一个应用程序，并在窗体上添加一个按钮，然后输入以下代码。见清单 15-3。

清单 15-3　检测是否为 Audio CD

```
//检测是否为音乐 CD，参数 Drive 表示检测的驱动器号
function IsAudioCD（Drive : char） : bool;
var
  DrivePath : string;
  MaximumComponentLength : DWORD;
  FileSystemFlags : DWORD;
  VolumeName : string;
begin
  Result := false;
  DrivePath := Drive + ':\';
  //检测是否为 CD-ROM
  if GetDriveType（PChar（DrivePath））  <> DRIVE_CDROM then exit;
  SetLength（VolumeName, 64）;
//检查卷标中是否带有 Audio CD，如果有表明该 CD 为 Audio CD
  GetVolumeInformation（PChar（DrivePath）,
                        PChar（VolumeName）,
                        Length（VolumeName）,
                        Nil,
                        MaximumComponentLength,
                        FileSystemFlags,
                        Nil,
                        0）;
  if lStrCmp（PChar（VolumeName）,'Audio CD'）  = 0 then result := true;
end;

//播放音乐 CD，参数 Drive 表示播放 CD 的驱动器符
function PlayAudioCD（Drive : char） : bool;
var
  mp : TMediaPlayer;
begin
  result := false;
  Application.ProcessMessages;
  if not IsAudioCD（Drive）  then exit;
  mp := TMediaPlayer.Create（Nil）;
  mp.Visible := false;
  mp.Parent := Application.MainForm;
```

```
    mp.Shareable := true;
    //指定设备类型为 dtCDAudio
    mp.DeviceType := dtCDAudio;
    mp.FileName := Drive + ':';
    mp.Shareable := true;
    mp.Open;
    Application.ProcessMessages;
    mp.Play;
    Application.ProcessMessages;
    mp.Close;
    Application.ProcessMessages;
    mp.free;
    result := true;
end;

procedure TForm1.Button1Click（Sender: TObject）;
begin
    //如果是音乐 CD，则播放
    if not PlayAudioCD（'D'） then
        ShowMessage（'Not an Audio CD'）;
end;
```

该代码中，定义了两个函数，第一个函数为 IsAudioCD（），用于测试是否为音乐 CD；第二个函数为 PlayAudioCD（），用于播放音乐 CD。

2. 查询 Win32 中 Audio CD 的自动执行特征及设置它

新建一个应用程序，并在窗体上放置两个按钮，然后输入以下代码，见清单 15-4。

清单 15-4　检测 Audio CD 自动执行选项及设置它

```
//从注册表中读取 Audio CD 的自动执行选项
function IsAudioCdAutoRunOn : bool;
var
    reg: TRegistry;
    Classes : string;
    ClassesRoot : string;
    i : integer;
begin
    reg := TRegistry.Create;
    reg.RootKey := HKEY_LOCAL_MACHINE;
    reg.OpenKey（'SOFTWARE\Classes\AudioCD\Shell',
                    false）;
```

```
Classes := reg.ReadString（''）；
reg.CloseKey;
reg.RootKey := HKEY_CLASSES_ROOT;
reg.OpenKey（'AudioCD\Shell',
                false）；
ClassesRoot := reg.ReadString（''）；
reg.CloseKey;
reg.free;
for i := 1 to length（Classes） do
  Classes[i] := UpCase（Classes[i]）；
for i := 1 to length（ClassesRoot） do
  ClassesRoot[i] := UpCase（ClassesRoot[i]）；
  result :=  （（Classes = 'PLAY'） and
              （ClassesRoot = 'PLAY'））；
end;

//设置自动执行选项
procedure SetAudioCdAutoRun（bOn : bool）；
var
  reg: TRegistry;
  Classes : string;
  ClassesRoot : string;
begin
  reg := TRegistry.Create;
  reg.RootKey := HKEY_LOCAL_MACHINE;
  reg.OpenKey（'SOFTWARE\Classes\AudioCD\Shell',
              false）；
  if bOn then
    reg.WriteString（'', 'play'） else
    reg.WriteString（'', ''）；
  reg.CloseKey;
  reg.RootKey := HKEY_CLASSES_ROOT;
  reg.OpenKey（'AudioCD\Shell',
              false）；
  if bOn then
    reg.WriteString（'', 'play'） else
    reg.WriteString（'', ''）；
  reg.CloseKey;
  reg.free;
end;
```

```
procedure TForm1.Button1Click（Sender: TObject）;
begin
  if IsAudioCdAutoRunOn then
    ShowMessage（'Autorun Audio Cd is On'） else
    ShowMessage（'Autorun Audio Cd is Off'）;
end;

procedure TForm1.Button2Click（Sender: TObject）;
begin
  SetAudioCdAutoRun（true）;
end;

procedure TForm1.Button3Click（Sender: TObject）;
begin
  SetAudioCdAutoRun（false）;
end;
```

上例中，通过读写注册表来读取和设置音乐 CD 的自动执行选项。

3. 获取 Audio CD 的序列号

新建应用程序，并在窗体中添加一个按钮以及 Tmemo 组件，然后输入以下代码，见清单 15-5。

清单 15-5　获取 Audio CD 序列号

```
uses MMSystem, MPlayer;

procedure TForm1.Button1Click（Sender: TObject）;
var
  mp : TMediaPlayer;
  msp : TMCI_INFO_PARMS;
  MediaString : array[0..255] of char;
  ret : longint;
begin
  mp := TMediaPlayer.Create（Nil）;
  mp.Visible := false;
  mp.Parent := Application.MainForm;
  mp.Shareable := true;
  mp.DeviceType := dtCDAudio;
  mp.FileName := 'D:';
  mp.Open;
```

```
    Application.ProcessMessages;
    FillChar（MediaString, sizeof（MediaString）, #0）;
    FillChar（msp, sizeof（msp）, #0）;
    msp.lpstrReturn := @MediaString;
    msp.dwRetSize := 255;
  //向多媒体设备发送 MCI_INFO 指令来获取信息
    ret := mciSendCommand（Mp.DeviceId,
                           MCI_INFO,
                           MCI_INFO_MEDIA_IDENTITY,
                           longint（@msp））;
    if Ret <> 0 then begin
     MciGetErrorString（ret, @MediaString, sizeof（MediaString））;
      Memo1.Lines.Add（StrPas（MediaString））;
    end else
       Memo1.Lines.Add（StrPas（MediaString））;
    mp.Close;
    Application.ProcessMessages;
    mp.free;
end;

end.
```



捕获视频图像



遇到难题

在数字监控以及可视电话中，需要将视频图像或者声音信息捕捉至计算机中，并进行数据的压缩、传输或者进行特殊的数据处理。例如进行汽车牌照的识别等等。在这个过程中，首要的是将视频数据捕捉至计算机中，然后需要将这些帧数据存成文件，以及将连续的声音和视频存成 avi 文件。如何实现这一技术呢？这一切是如此的令人兴奋。



钥匙在此

通过 AVICap 窗口类可以很容易地将视频捕捉能力加入到应用程序中。AVICap 为应用程序提供了一个简单、基于消息的接口来访问视频和波形音频硬件，并且控制捕捉流视频到磁盘的处理过程。

AVICap 支持实时的流视频捕捉和单帧的视频捕捉。另外，AVICap 还提供了对视频源（MCI 设备）进行控制的能力，如开始捕捉的位置以及停止位置等。

通过 AVICap 产生的 Windows 可以执行以下任务：

- 捕获音频流和视频流至 AVI 文件中；
- 动态地连接和断开视频与音频输入设备；
- 使用延迟或预览方法查看输入的视频信号；
- 在捕捉和拷贝捕捉的文件内容到另一个文件时，可以指定使用的文件；
- 设置捕捉的速率；
- 显示控制视频源和格式的对话框；
- 创建、存储和装载调色板；
- 拷贝图像和调色板至剪贴板中；
- 捕捉并存储图像作为一个设备独立位图（DIB）。

一个捕捉的基本过程包括：创建一个 AVICap 捕捉窗口；连接捕捉窗口与捕捉驱动程序；捕获数据；将捕获的数据存成文件；终止捕捉；断开捕捉窗口与捕捉驱动程序的连接。除了创建捕捉窗口之外，其余都是根据消息来进行操作的。

1. 创建捕捉窗口的函数—— capCreateCaptureWindow（）

该函数在 WinSDK 中是这样进行声明的：

```
HWND VFWAPI capCreateCaptureWindow（
    LPCSTR lpszWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWnd,
    int nID
）；
```

该函数用于创建一个捕捉窗口。其中，参数 lpszWindowName 载有用于捕捉窗口的名称；dwStyle 为捕捉窗口的类型，例如，WS_VISIBLE 表示窗口可见，具体可以参见 CreateWindowEx（）函数中的说明；x 和 y 为捕捉窗口的左上角的坐标；nWidth 和 nHeight 为捕捉窗口的宽度和高度；HWnd 为存放捕捉窗口的父窗口句柄；nID 为窗口的标识号。该函数的返回值为捕捉窗口的句柄。例如：

```
hWndC := capCreateCaptureWindowA（'My Own Capture Window',
                                  WS_CHILD or WS_VISIBLE ,
                                  Panel1.Left,
                                  Panel1.Top,
                                  Panel1.Width,
                                  Panel1.Height,
                                  Form1.Handle,
                                  0）;
```

2. 用于视频捕捉的常用的几个消息

WM_CAP_DRIVER_CONNECT 消息用于连接捕捉窗口与捕捉驱动程序；

WM_CAP_DRIVER_DISCONNECT 消息用于断开驱动程序与捕捉窗口的连接；

WM_CAP_GRAB_FRAME 用于从驱动程序中抓取和显示一帧；

WM_CAP_FILE_SAVEDIB 用于将当前的帧存成磁盘文件；

WM_CAP_FILE_SET_CAPTURE_FILE 用于为视频捕捉命名文件，该消息将文件名存储在内部结构中，但不会创建、分配和打开文件；

WM_CAP_SEQUENCE 初始化流视频和声音至文件中；

WM_CAP_STOP 终止捕捉操作。

例如，当要进行连接时，需要如下调用：

SendMessage（hWndC, WM_CAP_DRIVER_CONNECT, 0, 0）。



跟我来

要弄清一个完整的例子，可能您必须完全掌握微软的 Video for Windows SDK 的开发包。下面一个简化的例子显示了如何打开一个缺省的视频捕捉设备，并从设备上抓取视频的一帧数据，然后将这帧数据作为位图存成磁盘文件，并记录一个 AVI 文件（带有声音，但没有预览），最后关闭设备。注意，为了能使用此例，您的机器中必须装有视频捕捉设备，例如视频捕捉卡。

清单 15-6　视频捕捉程序

```
unit Unit1;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs, ExtCtrls, StdCtrls;

type
   TForm1 = class（TForm）
```

```
    Panel1: TPanel;
    OpenVideo: TButton;
    CloseVideo: TButton;
    GrabFrame: TButton;
    SaveBMP: TButton;
    StartAVI: TButton;
    StopAVI: TButton;
    SaveDialog1: TSaveDialog;
    procedure FormCreate（Sender: TObject）;
    procedure OpenVideoClick（Sender: TObject）;
    procedure CloseVideoClick（Sender: TObject）;
    procedure GrabFrameClick（Sender: TObject）;
    procedure SaveBMPClick（Sender: TObject）;
    procedure StartAVIClick（Sender: TObject）;
    procedure StopAVIClick（Sender: TObject）;
  private
    { Private declarations }
    hWndC : THandle;
    CapturingAVI : bool;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

const WM_CAP_START                    = WM_USER;
const WM_CAP_STOP                     = WM_CAP_START + 68;
const WM_CAP_DRIVER_CONNECT           = WM_CAP_START + 10;
const WM_CAP_DRIVER_DISCONNECT        = WM_CAP_START + 11;
const WM_CAP_SAVEDIB                  = WM_CAP_START + 25;
const WM_CAP_GRAB_FRAME               = WM_CAP_START + 60;
const WM_CAP_SEQUENCE                 = WM_CAP_START + 62;
const WM_CAP_FILE_SET_CAPTURE_FILEA = WM_CAP_START +  20;

function capCreateCaptureWindowA（lpszWindowName : PCHAR;
```

```
                                        dwStyle : longint;
                                        x : integer;
                                        y : integer;
                                        nWidth : integer;
                                        nHeight : integer;
                                        ParentWin   : HWND;
                                        nId : integer）: HWND;
                                        STDCALL EXTERNAL 'AVICAP32.DLL';

procedure TForm1.FormCreate（Sender: TObject）;
begin
  CapturingAVI := false;
  hWndC := 0;
  SaveDialog1.Options :=
    [ofHideReadOnly, ofNoChangeDir, ofPathMustExist]
end;

procedure TForm1.OpenVideoClick（Sender: TObject）;
begin
  //产生一个捕捉窗口，并返回该窗口句柄
  hWndC := capCreateCaptureWindowA（'My Own Capture Window',
                                        WS_CHILD or WS_VISIBLE ,
                                        Panel1.Left,
                                        Panel1.Top,
                                        Panel1.Width,
                                        Panel1.Height,
                                        Form1.Handle,
                                        0）;
  if hWndC <> 0 then
    SendMessage（hWndC, WM_CAP_DRIVER_CONNECT, 0, 0）;
    // WM_CAP_DRIVER_CONNECT 消息用于连接捕捉窗口与捕捉驱动程序
end;

procedure TForm1.CloseVideoClick（Sender: TObject）;
begin
  if hWndC <> 0 then begin
    SendMessage（hWndC, WM_CAP_DRIVER_DISCONNECT, 0, 0）;
   // WM_CAP_DRIVER_DISCONNECT 消息用于将驱动程序与捕捉窗口的连接断开
   hWndC := 0;
   end;
```

```
end;

//抓取一帧数据
procedure TForm1.GrabFrameClick（Sender: TObject）;
begin
  if hWndC <> 0 then
    SendMessage（hWndC, WM_CAP_GRAB_FRAME, 0, 0）;
// WM_CAP_GRAB_FRAME 用于从驱动程序中抓取和显示一帧数据
end;

procedure TForm1.SaveBMPClick（Sender: TObject）;
begin
  if hWndC <> 0 then begin
    SaveDialog1.DefaultExt := 'bmp';
    SaveDialog1.Filter := 'Bitmap files （*.bmp）|*.bmp';
    if SaveDialog1.Execute then
      SendMessage（hWndC,
                   WM_CAP_SAVEDIB,
                   0,
                   longint（pchar（SaveDialog1.FileName））;
    // WM_CAP_FILE_SAVEDIB 用于将当前的帧存成磁盘文件
  end;
end;

procedure TForm1.StartAVIClick（Sender: TObject）;
begin
  if hWndC <> 0 then begin
    SaveDialog1.DefaultExt := 'avi';
    SaveDialog1.Filter := 'AVI files （*.avi）|*.avi';
    if SaveDialog1.Execute then begin
      CapturingAVI := true;
      SendMessage（hWndC,
                   WM_CAP_FILE_SET_CAPTURE_FILEA,
                   0,
                   Longint（pchar（SaveDialog1.FileName））;
// WM_CAP_FILE_SET_CAPTURE_FILE 用于为视频捕捉命名文件，该消息将文件名
//存储在内部结构中，但不会创建、分配和打开文件
      SendMessage（hWndC, WM_CAP_SEQUENCE, 0, 0）;
// WM_CAP_SEQUENCE 初始化流视频和声音至文件中
    end;
```

```
    end;
end;

procedure TForm1.StopAVIClick（Sender: TObject）;
begin
    if hWndC <> 0 then begin
        SendMessage（hWndC, WM_CAP_STOP, 0, 0）;
        // WM_CAP_STOP 终止捕捉操作
        CapturingAVI := false;
    end;
end;
end.
```



制作屏幕保护程序



遇到难题

屏幕保护程序的使用在 Windows 中已经司空见惯了。而且，现在有很多商业软件在安装时就带了屏幕保护程序，以提高企业的形象。虽然屏幕保护程序是一个可执行文件，但相对于一般的应用程序来说，它所包含的功能或者说设计的目的又是不一样的。屏幕保护程序的编制与一般的应用程序的开发过程基本类似，不过又有其特殊的地方。



钥匙在此

一个屏幕保护程序拥有的功能应该包含预览、密码保护设置以及配置对话框等。先简单的复习一下屏幕保护程序是什么。若想更完整的了解，可以查询微软的 MSDN，或者求助于有关屏幕保护的站点或书籍。

首先，屏幕保护程序是一个可执行程序，它有一个扩展名为 SCR 而不是 EXE，在 Delphi 3 以上版本中可以使用$E 编译指令进行编译。

屏幕保护程序一般是通过以下几种方式启动：

- 当屏幕保护程序的启动时间到；
- 通过显示属性中“屏幕保护程序”页进行预览；
- 在设置该屏幕保护程序时进行测试预览；
- 通过全屏幕的预览；
- 通过改变屏幕保护程序的密码。

对于这几种方式，屏幕保护程序在执行时又是如何知道处于哪种方式的呢？或者说屏幕保护管理程序是如何通知屏幕保护程序的呢？屏幕保护程序可以带有不同的参数进行启动，因此屏幕保护程序是通过执行参数的传递知道现在处于哪种模式的。例如：

当启动屏幕程序的时间到时，屏幕保护程序的管理程序将带着参数/s、–s 或 s 来启动屏幕保护程序。在屏幕保护程序中可以通过 ParamStr（1）来获得这一参数；

进入显示属性的“屏幕保护程序”页中，屏幕保护程序则会先在小显示器中显示。此时，屏幕保护程序中 ParamStr（1）参数将载有/p、-p 或者只有 P，同时，ParamStr（2）中为小显示器的窗口句柄；

当您进行设置屏幕保护程序时，启动屏幕保护程序时根本无参数或者 ParamStr（1）为/C、/-C 或者 C；

当进行满屏预览屏幕保护程序时，这与时间到启动屏幕保护程序一样，ParamStr（1）带有/s 或–s 或 s 进行启动；

当您改变屏幕保护程序的密码时，ParamStr（1）的参数将带有/A、-A 或者只有 A。

在开发屏幕保护程序时应该注意，必须确保它在运行时不能被再次启动。另外，如果是通过显示属性进入屏幕保护程序的预览时，屏幕保护程序在小预览窗口中显示时会期待着两件事情，一是你必须等待，直到该窗口可见，二是当窗体不可见时，您必须关闭预览。

正如您所知道的一样，屏幕保护程序必须响应鼠标和按键事件。当没有密码时，屏幕保护程序应当就简单地关闭。当有密码设置时，它应该询问您的密码。

在创建屏幕保护程序之前的最后一个注意点就是，在调试屏幕保护程序时需要谨慎，确保在执行屏幕保护程序之前将源程序存储起来。因为一旦被锁住，您就有可能不得不重新启动，或者使用任务管理器将 Delphi 强制关闭。



跟我来

分别创建四个单元文件，名称为 ssetup.pas、ssave.pas、Globals.pas 和 CodeSpot.pas，其中前两个文件带有窗体文件。工程文件为 simple.dpr。完整代码参见清单 15-7。

清单 15-7　屏幕保护程序

```
//工程文件
program Simple;
```

```
uses
  Forms,
  SysUtils,
  Windows,
  Graphics,
  Classes,
  Ssave in 'SSave.pas' {Scrn},
  Codespot in 'CodeSpot.pas',
  Ssetup in 'SSetup.pas' {Setup},
  Globals in 'Globals.pas';
{$E SCR}
{$R *.RES}

var
  MySem          : THandle;
  Arg1,
  Arg2           : String;
  DemoWnd        : HWnd;
  MyRect         : TRect;
  MyCanvas       : TCanvas;
  x, y,
  dx, dy         : Integer;
  MyBkgBitmap,
  InMemBitmap : TBitmap;
  ScrWidth,
  ScrHeight      : Integer;
  SysDir         : String;
  NewLen         : Integer;
  MyMod          : THandle;
  PwdFunc        : function  (a : PChar; ParentHandle : THandle; b, c : Integer)  :
                   Integer; stdcall;

begin
  //获取屏幕程序的启动参数
  Arg1 := UpperCase (ParamStr (1));
  Arg2 := UpperCase (ParamStr (2));

  //根据不同的参数选择不同的显示方式
```

```
if （Copy（Arg1,1,2） = '/A'） or （Copy（Arg1,1,2） = '-A'） or
    （Copy（Arg1,1,1） = 'A'） then
  SSMode := ssSetPwd;

if （Copy（Arg1,1,2） = '/P'） or （Copy（Arg1,1,2） = '-P'） or
    （Copy（Arg1,1,1） = 'P'） then
  SSMode := ssPreview;

if （Copy（Arg1,1,2） = '/C'） or （Copy（Arg1,1,2） = '-C'） or
    （Copy（Arg1,1,1） = 'C'） or （Arg1 = ''） then
  SSMode := ssConfig;

//如果是密码设置方式,则启动密码设置对话框
if SSMode = ssSetPwd then begin
  SetLength（SysDir,MAX_PATH）;
  NewLen := GetSystemDirectory（PChar（SysDir）,MAX_PATH）;
  SetLength（SysDir,NewLen）;
  if （Length（SysDir） > 0） and （SysDir[Length（SysDir）] <> '\'） then
    SysDir := SysDir+'\';
  MyMod := LoadLibrary（PChar（SysDir+'MPR.DLL'））;
  if MyMod <> 0 then begin
    PwdFunc := GetProcAddress（MyMod,'PwdChangePasswordA'）;
    if Assigned（PwdFunc） then
      PwdFunc（'SCRSAVE',StrToInt（Arg2）,0,0）;
    FreeLibrary（MyMod）;
  end;
  Halt;
end;

MySem := CreateSemaphore（Nil,0,1,'SimpleSaverSemaphore'）;
if （（MySem <> 0） and （GetLastError = ERROR_ALREADY_EXISTS））then begin
  CloseHandle（MySem）;
  Halt;
end;

Application.Initialize;

//如果是预览方式时，则根据获得的第二个参数（小预览窗口的句柄）在小窗口中进
行预览
if SSMode = ssPreview then begin
```

```
DemoWnd := StrToInt（Arg2）;
while not IsWindowVisible（DemoWnd） do
  Application.ProcessMessages;
GetWindowRect（DemoWnd,MyRect）;
ScrWidth := MyRect.Right-MyRect.Left+1;
ScrHeight := MyRect.Bottom-MyRect.Top+1;
MyRect := Rect（0,0,ScrWidth-1,ScrHeight-1）;
MyCanvas := TCanvas.Create;
MyCanvas.Handle := GetDC（DemoWnd）;
MyCanvas.Pen.Color := clWhite;
x :=  （ScrWidth div 2）-16;
y :=  （ScrHeight div 2）-16;
dx := 1;
dy := 1;
MyBkgBitmap := TBitmap.Create;
with MyBkgBitmap do begin
  Width := ScrWidth;
  Height := ScrHeight;
end;
MyBkgBitmap.Canvas.FillRect（Rect（0,0,ScrWidth-1,ScrHeight-1））;
InMemBitmap := TBitmap.Create;
with InMemBitmap do begin
  Width := ScrWidth;
  Height := ScrHeight;
end;
//如果小窗口处于可见状态时,则在小窗口中画动画
while IsWindowVisible（DemoWnd） do begin
  //在指定背景的位置上画图 ,同时不停地更改绘制位置
  InMemBitmap.Canvas.CopyRect（MyRect,MyBkgBitmap.Canvas,MyRect）;
  InMemBitmap.Canvas.Draw（x,y,Application.Icon）;
  MyCanvas.CopyRect（MyRect,InMemBitmap.Canvas,MyRect）;
  Sleep（10）;
  Application.ProcessMessages;
  if  （x = 0）  or  （x =  （ScrWidth-33））  then
    dx := -dx;
  if  （y = 0）  or  （y =  （ScrHeight-33））  then
    dy := -dy;
  x := x+dx;
  y := y+dy;
end;
```

```
    MyBkgBitmap.Free;
    InMemBitmap.Free;
    MyCanvas.Free;
    CloseHandle（MySem）;
    Halt;
end;

//获取桌面位图
DesktopBitmap := TBitmap.Create;
with DesktopBitmap do begin
    Width := Screen.Width;
    Height := Screen.Height;
    end;
    BitBlt（DesktopBitmap.Canvas.Handle,0,0,Screen.Width,Screen.Height,
            GetDC（GetDesktopWindow）,0,0,SrcCopy）;

    //如果是配置方式，则启动配置窗体，否则启动预览的主窗体
    if SSMode = ssConfig then begin
      Application.CreateForm（TSetup, Setup）;
    end else
      Application.CreateForm（TScrn,Scrn）;

    Application.Run;

    DesktopBitmap.Free;
    CloseHandle（MySem）;
end.

{*************
*   SSetup.pas
*
* 该单元用于配置屏幕保护程序的参数
*
***************}

unit Ssetup;

interface

uses
```

```
    SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
    Forms, Dialogs, StdCtrls, ExtCtrls, Buttons, Spin, ExtDlgs;

type
    TSetup = class（TForm）
        Ball1Box: TGroupBox;
        Label3: TLabel;
        xPos1: TSpinEdit;
        yPos1: TSpinEdit;
        xVel1: TSpinEdit;
        yVel1: TSpinEdit;
        Label5: TLabel;
        Size1: TSpinEdit;
        Label7: TLabel;
        Label4: TLabel;
        Label8: TLabel;
        Random1: TCheckBox;
        OKButton: TBitBtn;
        CancelButton: TBitBtn;
        TestButton: TBitBtn;
        procedure TestButtonClick（Sender: TObject）;
        procedure FormCreate（Sender: TObject）;
        procedure FormActivate（Sender: TObject）;
        procedure OKButtonClick（Sender: TObject）;
        procedure CancelButtonClick（Sender: TObject）;
        procedure Random1Click（Sender: TObject）;
        procedure Size1Change（Sender: TObject）;
    private
        { Private declarations }
        Loading : Boolean;
    public
        { Public declarations }
    end;

var
    Setup: TSetup;

implementation

uses
```

```
    SSave, Globals;

{$R *.DFM}

procedure TSetup.TestButtonClick（Sender: TObject）;
begin
   DefRandom := Random1.Checked;
   DefSize := Size1.Value;
   DefPosX := xPos1.Value;
   DefPosY := yPos1.Value;
   DefVelX := xVel1.Value;
   DefVelY := yVel1.Value;
   TestMode := True;
   Scrn := TScrn.Create（Application）;
   Scrn.LoadingApp := True;
   Scrn.Left := -1000;
   Scrn.Top := -1000;
   Scrn.Width := 0;
   Scrn.Height := 0;
   Scrn.ShowModal;
   Scrn.Free;
   SetFocus;
   TestMode := False;
end;

procedure TSetup.FormCreate（Sender: TObject）;
begin
   Loading := True;
end;

procedure TSetup.FormActivate（Sender: TObject）;
begin
   if Loading then begin
      Loading := False;

      //从配置文件中读取参数值
      ReadIniFile;

      Size1.Value := DefSize;
```

```
    xPos1.Value := DefPosX;
    yPos1.Value := DefPosY;
    xVel1.Value := DefVelX;
    yVel1.Value := DefVelY;

    Random1.Checked := DefRandom;

    xPos1.MinValue :=  （DefSize*SpotSize div 2）+1;
    xPos1.MaxValue := Screen.Width-（DefSize*SpotSize div 2）;
    yPos1.MinValue :=  （DefSize*SpotSize div 2）+1;
    yPos1.MaxValue := Screen.Height-（DefSize*SpotSize div 2）;
end;

procedure TSetup.OKButtonClick（Sender: TObject）;
begin
    DefRandom := Random1.Checked;
    DefSize := Size1.Value;
    DefPosX := xPos1.Value;
    DefPosY := yPos1.Value;
    DefVelX := xVel1.Value;
    DefVelY := yVel1.Value;

    //将配置数据存入配置文件中
    WriteIniFile;
    Close;
end;

procedure TSetup.CancelButtonClick（Sender: TObject）;
begin
    Close;
end;

procedure TSetup.Random1Click（Sender: TObject）;
var
    NewColor : TColor;
begin
    NewColor := clWindow;
    with Random1 do begin
        if Checked then
```

```
        NewColor := clBtnFace;

    DefRandom := Checked;

    Size1.Enabled := not Checked;
    xPos1.Enabled := not Checked;
    yPos1.Enabled := not Checked;
    xVel1.Enabled := not Checked;
    yVel1.Enabled := not Checked;
end;

Size1.Color := NewColor;
    xPos1.Color := NewColor;
    yPos1.Color := NewColor;
    xVel1.Color := NewColor;
    yVel1.Color := NewColor;
end;

procedure TSetup.Size1Change（Sender: TObject）;
begin
    xPos1.MinValue :=  （Size1.Value*SpotSize div 2）+1;
    xPos1.MaxValue := Screen.Width-（Size1.Value*SpotSize div 2）;
    yPos1.MinValue :=  （Size1.Value*SpotSize div 2）+1;
    yPos1.MaxValue := Screen.Height-（Size1.Value*SpotSize div 2）;

    xPos1.Value := xPos1.Value;
    yPos1.Value := yPos1.Value;
end;

end.

{*************
*   SSave.pas
*
* 该单元为屏幕保护程序的主界面,用于处理在启动屏幕保护程序后所应该做的事情
*
***************}
unit Ssave;

interface
```

```
uses WinTypes, WinProcs, Graphics, Forms, Messages, Classes, Controls,
  ExtCtrls, StdCtrls, SysUtils;

type
  TScrn = class（TForm）
    Image1: TImage;
    procedure FormKeyDown（Sender: TObject; var Key: Word;
      Shift: TShiftState）;
    procedure FormCreate（Sender: TObject）;
    procedure FormMouseMove（Sender: TObject; Shift: TShiftState; X,
      Y: Integer）;
    procedure FormActivate（Sender: TObject）;
    procedure FormMouseDown（Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer）;
    procedure FormClose（Sender: TObject; var Action: TCloseAction）;
private
  { Private declarations }
  Mouse : TPoint;
  procedure StartSaver（var WinMsg : TMessage）; message WM_USER+1;
  procedure StopSaver（var WinMsg : TMessage）; message WM_USER+2;
  procedure GetPassword;
  procedure Trigger（Sender : TObject; var Done : Boolean）;
public
  { Public declarations }
  LoadingApp : Boolean;
end;

var
  Scrn : TScrn;
  DesktopBitmap : TBitmap;

implementation

uses
  CodeSpot, Globals, Registry;

const
  IgnoreCount : Integer = 0;
```

```
{$R *.DFM}

//关闭光标
procedure CursorOff;
begin
  ShowCursor（False）;
end;

//打开光标
procedure CursorOn;
begin
  ShowCursor（True）;
end;

//当接收到自定义的 WM_USER+1 消息时触发在桌面上画图的函数
procedure TScrn.StartSaver（var WinMsg : TMessage）;
begin
  DrawSpot;
end;

procedure TScrn.StopSaver（var WinMsg : TMessage）;
begin
  GetPassword;
end;

//从注册表中获取屏幕保护程序使用的密码
procedure TScrn.GetPassword;
var
  MyMod      : THandle;
  PwdFunc    : function  （Parent : THandle）  : Boolean; stdcall;
  SysDir     : String;
  NewLen     : Integer;
  MyReg      : TRegistry;
  OkToClose : Boolean;
begin
  if  （SSMode <> ssRun）  or TestMode then begin
    Close;
    Exit;
  end;
```

```
IgnoreCount := 5;
OkToClose := False;
MyReg := TRegistry.Create;
MyReg.RootKey := HKEY_CURRENT_USER;
if MyReg.OpenKey（'Control Panel\Desktop',False） then begin
  try
    try
      ShowCursor（True）;
      if MyReg.ReadInteger（'ScreenSaveUsePassword'） <> 0 then begin
        SetLength（SysDir,MAX_PATH）;
        NewLen := GetSystemDirectory（PChar（SysDir）,MAX_PATH）;
        SetLength（SysDir,NewLen）;
        if （Length（SysDir） > 0） and （SysDir[Length（SysDir）] <> '\'） then
          SysDir := SysDir+'\';
        MyMod := LoadLibrary（PChar（SysDir+'PASSWORD.CPL'））;
        if MyMod = 0 then
          OkToClose := True
        else begin
          PwdFunc := GetProcAddress（MyMod,'VerifyScreenSavePwd'）;
          if PwdFunc（Handle） then
            OkToClose := True;
          FreeLibrary（MyMod）;
        end;
      end
      else
        OkToClose := True;
    finally
      ShowCursor（False）;
    end;
  except
    OkToClose := True;
  end;
end
else
  OkToClose := True;

MyReg.Free;

if OkToClose then
  Close;
```

```
end;

procedure TScrn.Trigger（Sender : TObject; var Done : Boolean）;
begin
  PostMessage（Handle,WM_USER+1,0,0）;
end;

procedure TScrn.FormKeyDown（Sender: TObject; var Key: Word; Shift: TShiftState）;
begin
  GetPassword;
end;

procedure TScrn.FormMouseMove（Sender: TObject; Shift: TShiftState; X, Y: Integer）;
begin
  if IgnoreCount > 0 then begin
    Dec（IgnoreCount）;
    Exit;
  end;

  if （Mouse.X = -1） and （Mouse.Y = -1） then begin
    Mouse.X := X;
    Mouse.Y := Y;
  end
  else
    if （Abs（X-Mouse.X） > 2） and （Abs（Y-Mouse.Y） > 2） then begin
      Mouse.X := X;
      Mouse.Y := Y;
      GetPassword;
    end;
end;

procedure TScrn.FormCreate（Sender: TObject）;
begin
  LoadingApp := True;
end;

procedure TScrn.FormActivate（Sender: TObject）;
var
  Dummy : Boolean;
begin
```

```
if LoadingApp then begin
   LoadingApp := False;
   Scrn.Color := clBlack;
   Scrn.Top := 0;
   Scrn.Left := 0;
   Scrn.Width := Screen.Width;
   Scrn.Height := Screen.Height;
   InitSpot;
   Mouse.X := -1;
   Mouse.Y := -1;
   //当应用程序空闲时则运行 Trigger 函数
   Application.OnIdle := Trigger;
      SetWindowPos（Handle,HWND_TOPMOST,0,0,0,0,SWP_NOSIZE + SWP_NOM-
OVE）;
      SystemParametersInfo（SPI_SCREENSAVERRUNNING,1,@Dummy,0）;
      CursorOff;
      Scrn.Visible := True;
      SetCapture（Scrn.Handle）;
   end;
end;

procedure TScrn.FormMouseDown（Sender: TObject; Button: TMouseButton;
                                Shift: TShiftState; X, Y: Integer）;
begin
   GetPassword;
end;

procedure TScrn.FormClose（Sender: TObject; var Action: TCloseAction）;
var
   Dummy : Boolean;
begin
   SystemParametersInfo（SPI_SCREENSAVERRUNNING,0,@Dummy,0）;
   Application.OnIdle := Nil;
   ReleaseCapture;
   CursorOn;
end;

end.

{************
```

```
*   Codespot.pas
*
* 该单元用于在启动屏幕保护程序后在屏幕上绘图
*
***************}
unit Codespot;

interface

uses
  WinTypes, WinProcs, Graphics, Forms, Controls, Classes, Sysutils, Dialogs;

var
  zx, zy   : Integer;
  cx, cy,
  vx, vy,
  d        : Real;
  Picture : HBitmap;

procedure InitSpot;
procedure DrawSpot;

implementation

uses
  SSave, Globals;

procedure InitSpot;
begin
  Randomize;

  if not TestMode then
    ReadIniFile;

  zx := Screen.Width;
  zy := Screen.Height;

  d  :=  (Random (4) +1) *SpotSize;
  cx := Random ((zx div 2) -Round (d) -1) +1;
  cy := Random (zy-Round (d) -1) +1;
```

```
vx := Random（2）+1;
vy := Random（2）+1;
if Random（2） = 0 then
   vx := -vx;
if Random（2） = 0 then
   vy := -vy;

if not DefRandom then begin
   d := DefSize*SpotSize;
   cx := DefPosX-d/2;
   cy := DefPosY-d/2;
   vx := DefVelX;
   vy := DefVelY;
   end;

   Scrn.Image1.Picture.Bitmap := DesktopBitmap;
   Picture := Scrn.Image1.Picture.Bitmap.Handle;
end;

//绘制屏幕图像
procedure DrawSpot;
var
   WinDC, MemDC : HDC;
   Rgn1, Rgn3    : HRgn;
begin
   WinDC := GetDC（Scrn.Handle）;
   MemDC := CreateCompatibleDC（WinDC）;
SelectObject（MemDC,Picture）;

if  ((cx+vx <= 0)  or  (cx+d+vx >= zx))  then
   vx := -vx;

if  ((cy+vy <= 0)  or  (cy+d+vy >= zy))  then
   vy := -vy;

cx := cx+vx;
cy := cy+vy;

Rgn3 := CreateRectRgn（0,0,zx,zy）;
```

```
    Rgn1 := CreateEllipticRgn（Round（cx）,Round（cy）,
                                Round（cx+d）,Round（cy+d））;

    CombineRgn（Rgn3,Rgn3,Rgn1,RGN_DIFF）;
    FillRgn（WinDC,Rgn3,GetStockObject（BLACK_BRUSH））;

    SelectObject（WinDC,Rgn1）;
    BitBlt（WinDC,0,0,zx,zy,MemDC,0,0,SRCCOPY）;

    DeleteObject（Rgn3）;
    DeleteObject（Rgn1）;
    DeleteDC（MemDC）;
    ReleaseDC（Scrn.Handle,WinDC）;
end;

end.

{*************
*   Globals.pas
*
* 该单元用于读写屏幕保护程序的配置参数
*
***************}
unit Globals;

interface

type
   TSSMode =  （ssSetPwd,ssPreview,ssConfig,ssRun）;

const
   SSMode        : TSSMode = ssRun;

   TestMode      : Boolean = False;

   Section       = 'Screen Saver.Simple Screen Saver';

   SpotSize      = 50;
```

```
    DefSize      : Integer = 2;
    DefPosX      : Integer = 51;
    DefPosY      : Integer = 51;
    DefVelX      : Integer = 1;
    DefVelY      : Integer = 1;

    DefRandom    : Boolean = True;

procedure ReadIniFile;
procedure WriteIniFile;

implementation
uses
   IniFiles;

procedure ReadIniFile;
var
   IniFile : TIniFile;
begin
   IniFile := TIniFile.Create（'CONTROL.INI'）;

   DefSize := IniFile.ReadInteger（Section,'Size1',DefSize）;
   DefPosX := IniFile.ReadInteger（Section,'PosX1',DefPosX）;
   DefPosY := IniFile.ReadInteger（Section,'PosY1',DefPosY）;
   DefVelX := IniFile.ReadInteger（Section,'VelX1',DefVelX）;
   DefVelY := IniFile.ReadInteger（Section,'VelY1',DefVelY）;
   DefRandom := IniFile.ReadBool（Section,'Rand1',DefRandom）;

   IniFile.Free;
end;

procedure WriteIniFile;
var
   IniFile : TIniFile;
begin
   IniFile := TIniFile.Create（'CONTROL.INI'）;

   IniFile.WriteInteger（Section,'Size1',DefSize）;
```

```
    IniFile.WriteInteger（Section,'PosX1',DefPosX）;
    IniFile.WriteInteger（Section,'PosY1',DefPosY）;
    IniFile.WriteInteger（Section,'VelX1',DefVelX）;
    IniFile.WriteInteger（Section,'VelY1',DefVelY）;

    IniFile.WriteBool（Section,'Rand1',DefRandom）;

    IniFile.Free;
end;

end.
```



图 15-1　屏幕保护设置对话框

先编译该屏幕保护程序，并将编译的屏幕保护程序 Simple.scr 拷贝到系统目录中，例如 C:\WinNT\System32 或 C:\Win95\System。然后，启动显示属性的屏幕保护程序页面，选择自己的屏幕保护程序，接着便可以试试您的保护程序了。在屏幕保护程序的设置时，可以看到如图 15-1 所示的结果。



打开和关闭屏幕保护程序



遇到难题

上一节介绍了如何制作屏幕保护程序。屏幕保护程序虽然是用于保护我们的显示设备的，但有时这些屏幕保护程序也会带来麻烦。例如，在观看 VCD 时，由于长时间无键盘以

及鼠标输入，于是到了一定的时间后，计算机会自动启动屏幕保护程序（当然这是在您已经使用了屏幕保护程序的前提下），从而给观看 VCD 带来了麻烦，此时应当关闭掉这个屏幕保护程序或者延期使用屏幕保护程序。这可以通过编一个小程序来控制屏幕保护程序的打开或关闭。



钥匙在此

要实现屏幕保护程序的打开和关闭，需要使用 Win32 API 函数 SystemParametersInfo（）来读取和设置系统的参数。该函数在 Windows.pas 中是如下声明的：

```
function SystemParametersInfo（uiAction, uiParam: UINT;
  pvParam: Pointer; fWinIni: UINT）: BOOL; stdcall;
```

其中，uiAction 参数指定了要查询和设置的参数名称，uiParam 和 pvParam 参数的意义完全取决于 uiAction 参数的种类。如果系统参数被修改，由 fWinIni 参数确定用户的外形（profile）是否被修改，以及修改后是否使用 WM_SETTINGCHANGE 消息广播到当前打开的所有顶层窗口。uiAction 系统参数的名称值较多，可以在 Win32 API 的 SystemParametersInfo（）函数中查阅，此处只介绍与本节有关的系统参数，例如：

SPI_SETSCREENSAVEACTIVE 参数

该系统参数为屏幕保护设置参数，用于设置屏幕保护程序的状态。如果 uiParam 指定为 True 时表示打开屏幕保护程序，否则表示关闭屏幕保护程序，例如执行下列语句将关闭屏幕保护程序。

```
SystemParametersInfo（SPI_SETSCREENSAVEACTIVE,
                      0,
                      Nil,
                      0）;
```



跟我来

使用下列代码可以简单地进行屏幕保护程序的打开和关闭，见清单 15-8。

清单 15-8　打开和关闭屏幕保护程序

```
procedure TForm1.Button1Click（Sender: TObject）;
begin
{Turn it off}
  SystemParametersInfo（SPI_SETSCREENSAVEACTIVE,
                        0,
                        Nil,
                        0）;
```

```
{Turn it on}
  SystemParametersInfo（SPI_SETSCREENSAVEACTIVE,
                        1,
                        Nil,
                        0）;
end;
```

第十六章

DBGrid 高级应用

热点透视

DBGrid 是 Delphi 中用于显示数据库数据的高级组件，其功能比较强大。DBGrid 是一个数据感应控件，在一般应用中使用的是该控件的缺省选项。这使得该控件的极强的开发性没有得到充分应用。本章将重点放在提高对 DBGrid 的应用水平上，使其能够发挥更大的用处。通过这些应用一方面可以使用户录入数据时更为方便，另一方面加强了该控件的显示功能。本章内容主要包括：

- 实现 DBGrid 的下拉列表框输入
- 为 DBGrid 插入组件
- 改变 DBGrid 单元格的缺省颜色
- 在 DBGrid 中实现拖拉技术



实现 DBGrid 的下拉列表框输入



遇到难题

我们当然希望 DBGrid 能够在数据录入时支持下拉列表框的输入。例如，您有两张表，一张包括一些书信息，另一张包括一些作者信息。如果您需要向 Book 表中增加一本新书时，若能自动看到一列存在的作者，将会使录入的工作轻松很多。那样，您可以只用输入书名，然后通过选择下拉列表框实现作者的录入，这样的录入工作会快很多。Book 表将会自动把 Auther 表里相应作者的说明包括在内。也就是说，Author 表中的作者号会自动插入 Book 表中。那么在 Delphi 中如何实现呢？



钥匙在此

可以通过 Delphi 中的查找字段技术在一个表中查找另一个标中使用的数据。

什么叫查找字段？先举一个例子给以说明。

假如现在有两张表，表名分别为 Book.db 和 Author.db。前者为书的信息表，后者为作者信息表。Book 表结构说明如表 16-1 所示，Author 表结构如表 16-2 所示。

表 16-1　　Book 表结构

字段名称	数据类型	长度	备注
书号	字符	20	关键字
书名	字符	40	
作者编号	整数	4	

表 16-2　　Author 表

字段名称	数据类型	长度	备注
作者编号	整数	4	关键字
作者	字符	8	

假设作者信息表已经全部录入完了，现在正往 Book 表输入记录，在输入作者编号时希望显示的是作者的名字而不是作者的编号。但在设计数据库时，一方面为了节省空间，另一方面为了减少数据的冗余，所以在 Book 表的结构中只有作者的编号。因此，为了在录入 Book 表的作者编号时能够直接从另一张表中查询或选择，这样的字段即为查找字段（Lookup Field）。同时，通过定义查找字段还可以进行一些信息的关联，例如，在录入作者的编号时，

希望直接显示的是作者的名字，而写入数据库的又是作者的编号。这样会给数据的录入带来极大的方便。



跟我来

下面举一个实际的例子来说明如何设置和使用查找字段。

现在以表 16-1 和表 16-2 作为应用程序的表结构，并且假设作者库中已经输入了以下三个作者的信息。

1	季雪刚
2	张宏林
3	李四

第一步：创建应用程序，并在主窗体上增加两个 Ttable 控件、TDBGrid 以及 TDBNavigator 控件。

第二步：设置相关的数据库属性。

```
Table1.databasename:='mydata';//表示表所存放的数据库别名
Table2.databasename:='mydata';
Table1.tablename:=Book.db;
Table2.tablename:=Author.db;
Datasoure1.dataset:=table1;
DBGrid1.datasource:=datasoure1;
DBNavigator1.datasource:=datasource1;
```

第三步：为 Table1 添加查找字段。首先选择 table1，然后点击鼠标右键选择菜单“Fields Editor...”，并在该对话框上选择右键菜单“add all fields”，接着在字段编辑器“Fields Editor”中加入所有字段。结果如图 16-1 所示。

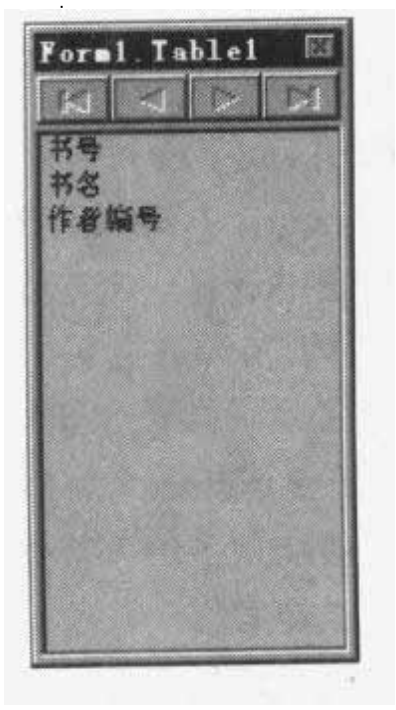


图 16-1　Fields Editor 对话框

在该对话框中点击鼠标右键，然后选择“new field”菜单，此时会弹出新字段定义对话框，如图 16-2 所示。

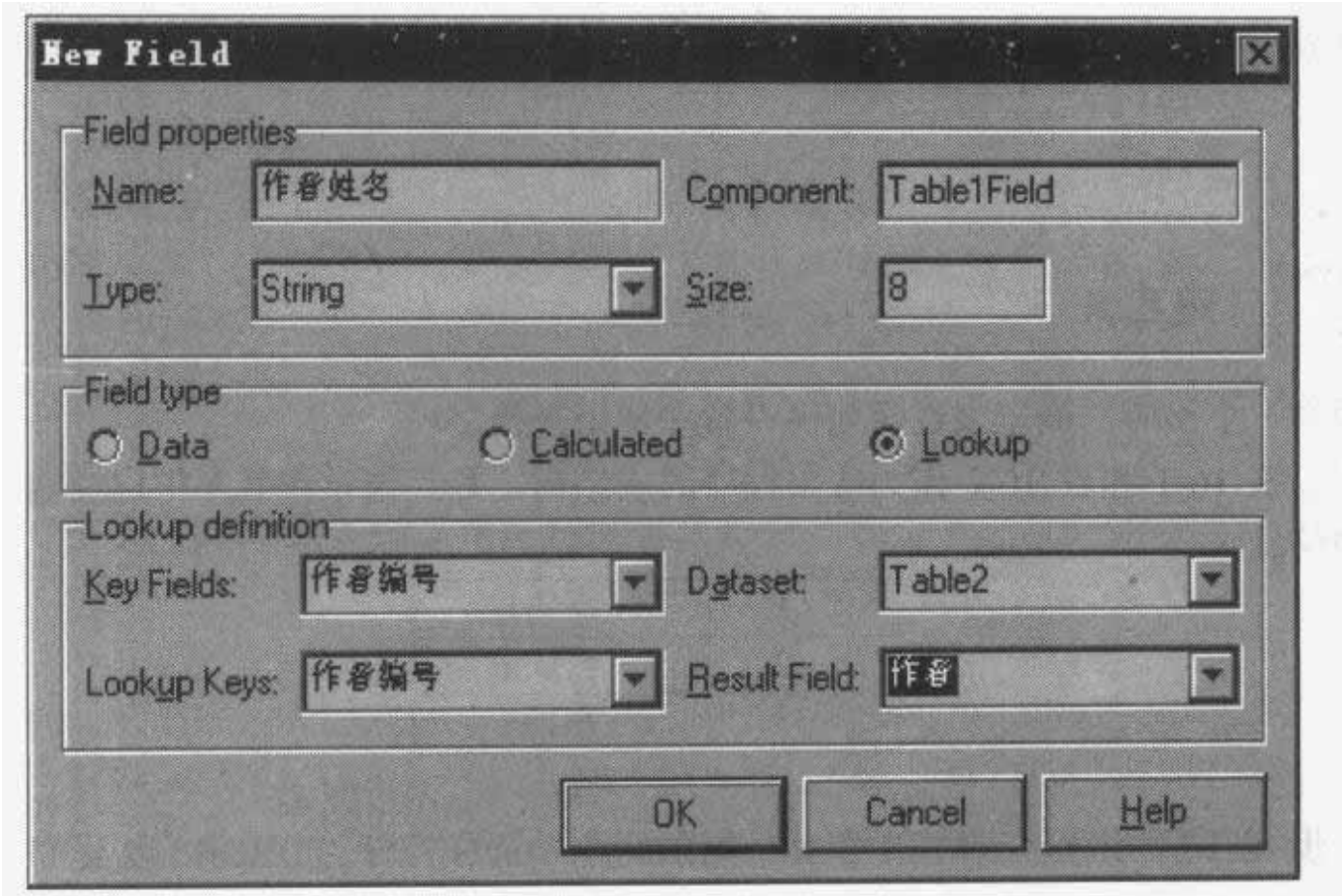


图 16-2　定义查找字段

接着是对新定义的字段进行设置的过程：

（1）为查找字段在“Name”编辑框中敲入名称。注意不要敲入已经存在的字段名。本例为“作者姓名”。

（2）从“Type”组合框中选择字段的数据类型。本例为 string。

（3）在 Size 编辑框中输入字段的大小。大小只在数据类型为 TstringField、TBytesField 和 TvarBytesField 时才有效。本例设置为 8。

（4）在“Field Type”选择框中选择“Lookup”类型。选择后会使“Dataset”以及“Key Fields”组合框有效。

（5）从“Dataset”组合框中选择要使用的数据集。注意该数据集不能选择自己，应该选择其他数据集，否则会产生循环异常。指定该数据集后会使“Lookup keys”和“Result Field”组合框有效。本例选择 Table2。

（6）从“key fields”的下拉列表中选择一个用于与另一个表相匹配的字段（该字段为当前数据集的字段）。为了指定更多的匹配字段可以直接输入字段名，而不是通过选择下拉列表来实现，各字段名之间通过分号隔开。本例只选择一个字段为“作者编号”。

（7）从“Lookup keys”下拉列表中选择一个与源数据集中所选择的字段相匹配的字段（该字段为查找数据集的字段），即用于与源数据集中的字段进行关联的字段。为了指定更多的字段，可以直接输入相应的字段名，同样通过分号隔开。本例为“作者编号”。

（8）从“Result Field”下拉列表中选择一个字段（为查找数据集中的字段）作为查找字段返回值，即为查找字段。如果需要从更多的字段返回值的话，可以直接指定相应的字段名，同样通过分号分开。本例选择了“作者”。

注意：如果设置多个字段的话，应当使每个字段相对应。

第四步：激活 Table1 组件，并运行程序。

运行结果如图 16-3 所示。图中为在录入“C++Builder”一书时从作者的下拉框中选择作者名单时的情况。而真正写入 book.db 数据库中的则为该作者的代号。



图 16-3　运用查找字段进行数据录入



为 DBGrid 插入组件



遇到难题

Delphi 提供了功能强大的 DBGrid 组件，以方便数据库应用程序的设计。但是如果我们仅仅利用 DBGrid 组件，获得焦点网格（Grid）的只是一个简单的文本编辑框或者是下拉列表框（这在“如何实现 DBGrid 的下拉列表框输入”一节已经介绍过了），不方便用户输入数据。Delphi 也提供了一些其他数据组件来方便用户输入，比如 DBComboBox 以及 DBCheckBox 等组件，但这些组件没有 DBGrid 功能强大。Delphi 能不能像 Visual Foxpro 那样让 DBGrid 中获得焦点网格的可以是其他可视数据组件以方便用户呢？



钥匙在此

其实可以通过在 DBGrid 中插入其他可视组件来解决这一问题。因为 Delphi 提供了重新绘制网格的机制，所以可以使用 TCustomDBGrid.OnDrawDataCell 事件书写句柄来绘制您的数据。

Delphi 对 DBGrid 处理的内部机制可以这样来理解，当通过 DBGrid 进行数据输入时，便会在网格上浮动一个 DBEdit 组件。你输入数据的网格其实是浮动的 DBEdit 组件，而其他未获得焦点网格的地方不过是绘制的图像罢了。因此，为了能在网格中绘制自己的组件，可以通过在网格上浮动一个可视组件来实现。这样的话，从简单的 DbCheckBox 到复杂的对话框，都可以在 DBGrid 中插入。下面就是如何在 DBGrid 中插入 TDBLookupComboBox 组

件的步骤，采用同样的办法可以插入其他组件。



跟我来

本例将模仿前一个章节的功能，来实现 DBGrid 下拉列表框的输入。

第一步：创建应用程序，然后将两个 TDataSource、两个 Ttable、一个 TDBGrid、一个 TDBLookupComboBox 以及一个 TDBNavigator 组件放置到 Form1 上。

第二步：为组件设置属性。

```
Table1
DatabaseName :='MyData';
TableName :='Book.db';
    Table2
DatabaseName :='MyData';
TableName :='Author.db';
    Datasource1.Dataset:=Table1;
        Datasource2.Dataset:=Table2;
        DBNavigator.Dataset:=Table1;
        DBGrid1.Dataset:=Table1;
        DBLookupComboBox1
            Datasource:=Datasource1;
            DataField:= '作者编号';
        ListSource:=Datasource2;
        KeyField:= '作者';
        ListField:= '作者编号';
```

第三步：书写 DrawDataCell 事件例程。DrawDataCell 事件用来重新绘制单元格。当获得焦点的网格所对应的字段与 DBLookupComboBox1 所对应的字段一致时，便移动 DBLookupComboBox1 到获得焦点的网格上，并且使组合框可视，从而实现在 DBGrid 的指定列上显示 DBLookupComboBox1 的功能。设置 DBGrid1 的 DrawDataCell1 事件如下：

```
procedure TForm1.DBGrid1 DrawDataCell
(Sender: TObject; const Rect: TRect; DataCol: Integer; Column: TColumn; State:
TGridDrawState);
begin
  if (gdFocused in State) then
  begin
    with DBLookupComboBox1 do
      if (Field.FieldName = DataField ) then
      begin
          Left := Rect.Left + DBGrid1.Left;
```

```
            Top := Rect.Top + DBGrid1.top;
            Width := Rect.Right - Rect.Left;
            Height := Rect.Bottom - Rect.Top;
            Visible := True;
        end;
    end;
end;
```

第四步：书写 DBGrid1 的 OnColExit 事件例程。规定单元格在未获得焦点时不显示 DBLookupComboBox1，设置 DBGrid1 的 OnColExit 事件如下：

```
procedure TForm1.DBGrid1ColExit(Sender: TObject);
begin
  If DBGrid1.SelectedField.FieldName
  = DBLookupComboBox1.DataField then
  begin
      DBLookupComboBox1.Visible := false;
  end;
end;
```

第五步：书写 DBGrid1 的 KeyPress 例程。当 DBGrid 的指定列获得焦点时，DrawDataCell 事件只是绘制单元格，并显示 DBLookupComboBox1，此时 DBLookupComboBox1 并没有获得焦点，于是数据的输入还是在单元格上进行。但这时可以在 DBGrid1 的 KeyPress 事件中调用 SendMessage 这个 Windows API 函数将数据输入传输到 DBLookupComboBox1 上，从而达到在 DBLookupComboBox1 上进行数据输入的目的。因此要设置 KeyPress 事件如下：

```
procedure TForm1.DBGrid1KeyPress
(Sender: TObject; var Key: Char);
begin
  if (key < > chr(9)) then
    begin
      if (DBGrid1.SelectedField.FieldName
      = DBLookupComboBox1.DataField) then
      begin
            DBLookupComboBox1.SetFocus;
            SendMessage(DBLookupComboBox1.Handle,
        WM_Char, word(Key), 0);
      end;
    end;
end;
```

第六步：在 Form1.show 事件中加入以下代码，以将 DBLookupComboBox1 初始化为隐藏状态。

```
procedure TForm1.FormShow(Sender: TObject);
begin
    DBLookupComboBox1.Visible:=false;
end;
```

第七步：激活 Table1 和 Table2。运行程序，运行结果如图 16-4 所示。

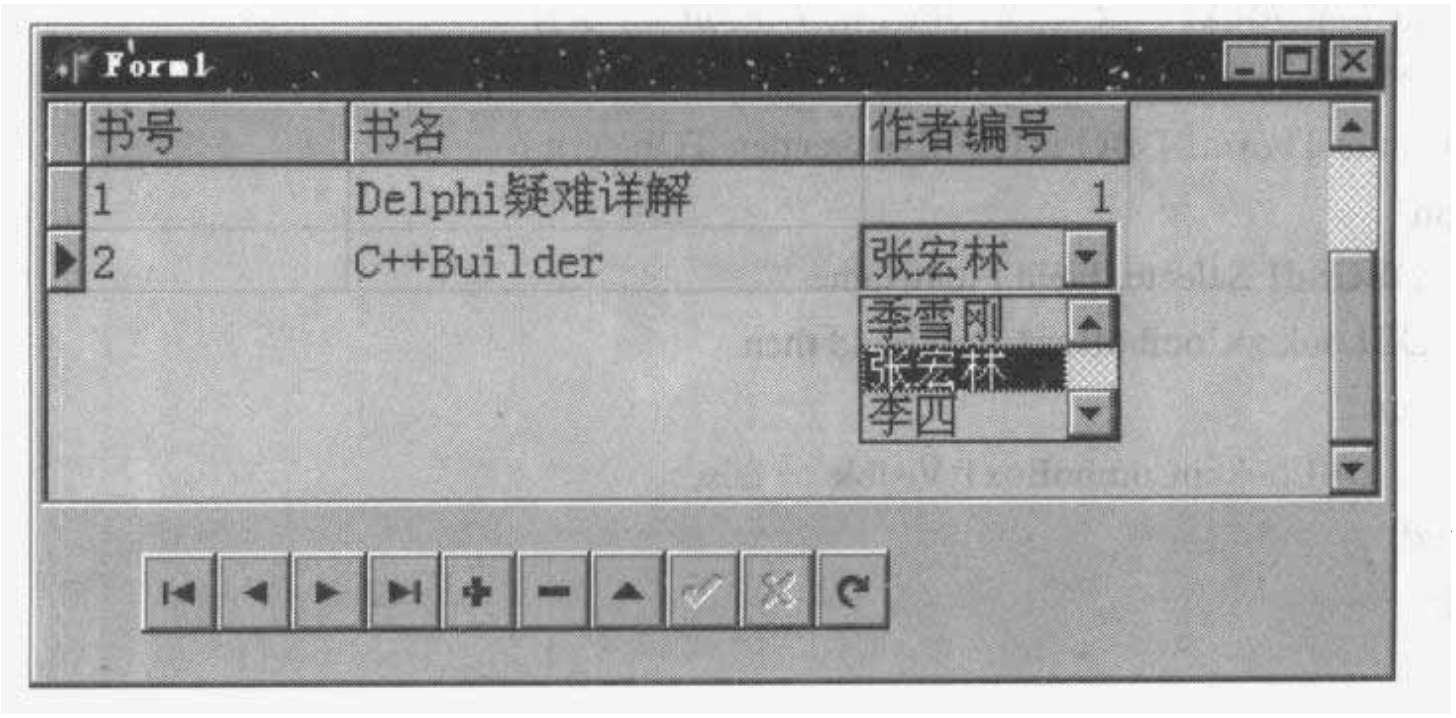


图 16-4　向 DBGrid 中插入组件

比较图 16-3 和图 16-4 不难看出，前者的每一行都为作者名称，而后者只有在焦点集中到作者编号时才显示相应的作者的名称。所以，如果是想实现下拉列表框的输入时，作者建议使用前一节介绍的方法；如果想动态地为 DBGrid 添加控件时则使用本节所介绍的方法。



改变 DBGrid 单元格缺省的颜色



遇到难题

在编程中改变 DBGrid 的缺省颜色是经常发生的。如在图书借阅管理系统中，统计时需要特殊地显示那些过期未还者的记录。又如，在借款中需要对那些超支的借方记录进行特殊地显示。我们知道，在 DBGrid 中未选中记录的缺省颜色为黑色，选中记录为深蓝色。而且，在 DBGrid 中更改字体颜色时是针对所有记录，因此，如果要对特殊的记录用不同颜色显示时需要通过编程来实现。



钥匙在此

为了能在程序中根据一定的条件动态地改变 DBGrid 中文本项内容的颜色，于是 Delphi 提供了动态更改单元格的机制。这首先需要截获绘制单元格的事件，然后根据您提供的条件来更改绘制单元格的缺省值，包括颜色。

绘制单元格的事件声明如下：

```
Type
  TDrawColumnCellEvent = procedure (Sender: TObject; const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState) of object;//函数过程说明
  property OnDrawColumnCell: TDrawColumnCellEvent;
```

该事件是在网格（grid）需要绘制单元格（cell）时激发的。也正是该事件为程序员提供了更改单元格数据的显示形式的机会。可以使用 Canvas 属性提供的方法来绘制单元格。

其中，参数 Rect 表示在画布上的位置；参数 DataCol 是当前单元格所在列在 Columns 数组中的索引；Column 参数则是描述显示属性和字段与单元格绑定关系的 Tcolumn 对象；State 参数指定单元格是否具有输入焦点、是否被选择以及是否为固定（非数据区）单元格等，例如某列头。

在 OnDrawColumnCell 事件例程中，可以调用 DefaultDrawColumnCell 方法来通知对数据感应的网格控件 DBGrid 将数据写入当前单元格中。OnDrawColumnCell 事件处理可以增强 DefaultDrawColumnCell 方法提供的可视选择和焦点指示的功能。

如果将 DefaultDrawing 设为 True，则数据在 OnDrawColumnCell 事件发生之前就会在单元格中输出，并且该网格在 OnDrawColumnCell 事件之后将在所选择的单元格周围绘制一个焦点矩形。



如果 Columns 属性有一个 csDefault 缺省的 State 值，则 OnDrawDataCell 事件在 OnDrawColumnCell 事件之前就会发生。

在单元格中绘制文本的函数为 DefaultDrawColumnCell（），该函数的声明格式如下：

```
procedure DefaultDrawColumnCell(const Rect: TRect; DataCol: Integer; Column: TColumn; State: TGridDrawState);
```

功能描述：在 OnDrawColumnCell 事件处理中，调用 DefaultDrawColumnCell 来查找字段的表示值并且将之写到单元格中。当 DefaultDrawing 为真时，DefaultDrawColumnCell 函数除了不绘制所选择的焦点矩形外，会做网格控件所做的任何事。

Rect 参数表示单元格在画布上的位置；DataCol 参数是表示列在 Columns 数组中的索引值；Column 参数则是描述显示属性和字段与单元格绑定关系的 Tcolumn 对象；State 参数指定单元格是否具有输入焦点。是否被选择以及是否为固定（非数据区）单元格等，例如某列头。

下面将根据给定条件来改变不同行的数据颜色和所选行的颜色。



跟我来

本例所使用的表结构描述如下：

表名为 borrow.db

表结构：借阅人 应归还日期

本例将完成两个任务，一个是将所有已经过期（即已经超过了当前日期）的记录用红色显示，而当前选择的记录行以黄色显示。由于本例将通过 DBGrid 进行数据编辑，所以不是通过设置控件的 dgRowSelect 选项来选择行的。因为需要对 DBGrid 进行数据编辑，所以需要通过重新绘制单元格来实现行的选择。

新建应用程序，然后为该窗体添加 Ttable、Tdatasource、Tdbgrid 以及 tdbnavigator 等控件并设置相应的属性。其源代码如清单 16-1 所示。

清单 16-1　更改 DBGrid 缺省颜色

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, DBTables, ExtCtrls, DBCtrls, Grids, DBGrids;

type

  TCustomDBGridCracker = class(TCustomDBGrid);

  TForm1 = class(TForm)
    DBGrid1: TDBGrid;
    DBNavigator1: TDBNavigator;
    Table1: TTable;
    DataSource1: TDataSource;
    procedure DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
      DataCol: Integer; Column: TColumn; State: TGridDrawState);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.DBGrid1DrawColumnCell(Sender: TObject; const Rect: TRect;
  DataCol: Integer; Column: TColumn; State: TGridDrawState);
begin
    //为了使用 DataLink 属性，自定义了一个类，使其能引用该属性
    with TCustomDBGridCracker(sender) do begin
    //判断当前行是否被选中，若被选中则将颜色变成黄色
    if DataLink.ActiveRecord = Row - 1 then
      Canvas.Brush.Color := clYellow
    else
    //判断该借阅人是否已经超期了，如果是则将颜色变成红色
    if table1.FieldByName('应归还日期').asdatetime>date then
      Canvas.Brush.Color := clred
    else
      Canvas.Brush.Color := clwhite;
    //根据上面设置的颜色值绘制单元格
    DefaultDrawColumnCell(Rect, DataCol, Column, State);
  end;

end;

end.
```

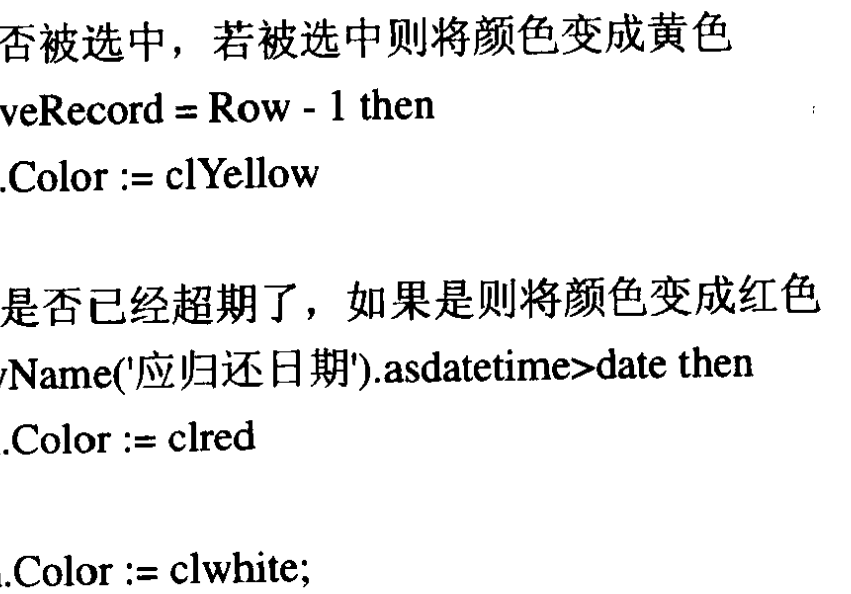


图 16-5　更改 DBGrid 颜色

运行该程序后，结果如图 16-5 所示，其中的“李四”颜色为黄色，表示当前选中的记录。借阅人为季雪刚的因为未过期而显示为白色，而张宏林借阅过期（与当地时间 2000 年一月十一日比较）而变成红色了。



在 DBGrid 中实现拖拉技术



遇到难题

在数据库中经常会碰到主从表的设计，如一个团组信息表和一张个人信息表，其中个人从属于团组。有时个人经常会出现团组间的调配，这样就使得数据的从属关系也发生变化。因此，为了输入的简单化，应该考虑通过拖拽的功能来实现这一输入，就如同文件在文件夹之间的移动一样。要实现这样的输入方法，首先要解决的就是实现 DBGrid 间的拖拽技术。



钥匙在此

使用拖放技术极大地增加了用户界面的复杂性。

一般而言，拖拉技术需要程序能实现四个可能的用户动作：

（1）开始一个拖动操作；

（2）接受一个被拖放的项；

（3）放下项；

（4）结束一个拖动操作。

对 DBGrid 的托拽也类似。首先应当设置拖放模式 DragMode。可以将其设为自动方式 dmAutomatic，即只要用户进行拖放时即可允许拖放的进行。如果将其设成 dmManul 手工方式时，则程序可以控制它在什么时候允许拖放操作。

接着，处理被拖放部件的回执，即把：

Accept:=True;

添加到被拖放部件的 OnDragOverEvent 中。

该操作的第三步是在接受方的 OnDragDrop 事件中完成的。

最后，如果愿意的话可以通知用户拖放已经结束了，此时可以在被拖动的控件的 OnDragEnd 事件中添加相应的代码。



跟我来

下例将演示通过简单的托拽方式来实现两个 DBGrid 之间数据的移动。

1. 创建一个简单组件

创建的组件名称为 TMyDBGrid，其组类为 TDBGrid。该控件的定义代码如清单 16-2 所示。

清单 16-2　创建 TMyDBGrid 控件组

```
unit myDBGrid;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Grids, DBGrids;

type
  TmyDBGrid = class(TDBGrid)
  private

    { Private declarations }
    FOnMouseDown: TMouseEvent;
  protected
    { Protected declarations }
    procedure MouseDown(Button: TMouseButton; Shift: TShiftState;
      X, Y: Integer); override;
  published
    { Published declarations }
    property Row;
    property OnMouseDown read FOnMouseDown write FOnMouseDown;
  end;

procedure Register;

implementation

procedure TMyDBGrid.MouseDown(Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

```
begin
   if Assigned(FOnMouseDown) then
      FOnMouseDown(Self, Button, Shift, X, Y);
   inherited MouseDown(Button, Shift, X, Y);
end;

procedure Register;
begin
   RegisterComponents('Samples', [TMyDBGrid]);
end;

end.
```

然后将该组件安装到 Delphi 的控件面板上。

2. 创建基于上述组件的应用程序，其源代码如清单 16-3 所示。

清单 16-3　实现 DBGrid 间的托拽

```
unit GridU1;

interface

uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
   Dialogs, Db, DBTables, Grids, DBGrids, MyDBGrid, StdCtrls;

type
   TForm1 = class(TForm)
      MyDBGrid1: TMyDBGrid;
      Table1: TTable;
      DataSource1: TDataSource;
      Table2: TTable;
      DataSource2: TDataSource;
      MyDBGrid2: TMyDBGrid;
      procedure MyDBGrid1MouseDown(Sender: TObject;
         Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
      procedure MyDBGrid1DragOver(Sender, Source: TObject;
         X, Y: Integer; State: TDragState; var Accept: Boolean);
      procedure MyDBGrid1DragDrop(Sender, Source: TObject;
         X, Y: Integer);
   private
```

```
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

var
  SGC : TGridCoord;

procedure TForm1.MyDBGrid1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  DG : TMyDBGrid;
begin
  DG := Sender as TMyDBGrid;
  SGC := DG.MouseCoord(X,Y);
  if (SGC.X > 0) and (SGC.Y > 0) then
    (Sender as TMyDBGrid).BeginDrag(False);
end;

procedure TForm1.MyDBGrid1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
var
  GC : TGridCoord;
begin
  GC := (Sender as TMyDBGrid).MouseCoord(X,Y);
  Accept := Source is TMyDBGrid and (GC.X > 0) and (GC.Y > 0);
end;

procedure TForm1.MyDBGrid1DragDrop(Sender, Source: TObject;
  X, Y: Integer);
var
  DG      : TMyDBGrid;
  GC      : TGridCoord;
```

```
    CurRow : Integer;
  begin
    DG := Sender as TMyDBGrid;
    GC := DG.MouseCoord(X,Y);
    with DG.DataSource.DataSet do begin
      with (Source as TMyDBGrid).DataSource.DataSet do
        Caption := 'You dragged "'+Fields[SGC.X-1].AsString+'"';
      DisableControls;
      CurRow := DG.Row;
      MoveBy(GC.Y-CurRow);
      Caption := Caption+' to "'+Fields[GC.X-1].AsString+'"';
      MoveBy(CurRow-GC.Y);
      EnableControls;
    end;
  end;

  end.
```

第十七章

快速报表

热点透视

报表在一些信息管理系统的程序中占有很重要的位置，因此，能否快速地设计出较理想的报表直接关系到程序实用化程度的高低。本章一方面介绍了创建报表的基础知识，另一方面又详细的讨论了如何扩展 QuickReport 功能。相信通过本章的学习可以掌握报表设计的基础知识以及开发报表控件的高级技术。本章内容主要包括：

- 快速创建报表
- 创建自定义模板
- 进行 QuickReport 附件的开发



快速创建报表



遇到难题

报表是直接体现用户工作量的内容之一，所以报表制作的好坏直接影响到用户对软件的使用率。Delphi 中提供了很好的快速创建报表的工具和控件，称之为快速报表。

快速报表是一组构件包，存在于 Delphi 构件库面板的 Qreport 页面上。快速报表是一个报表生成器，是第三方提供的专门制作报表的构件包。该构件包的制造商为 Qusoft 公司，用户可以访问网站http://www.qusoft.com，以获取最新的版本信息或者资源，现在的 QuickReport 版本为 3.x。它给程序员提供了快速生成不同复杂程度的报表的能力。

那么如何能快速地掌握这个报表生成器呢？



钥匙在此

为了能很快地掌握这一工具，我们首先应当从以下几个方面去着手。

1. 报表的组成

报表定义了数据输出的外观，因此，在进行报表设计之前，应该清楚报表是由哪几部分构成的。

图 17-1 为一个较为典型的简单报表，这是一个顾客列表，主要有公司、联系人、电话、传真和编号等内容，该报表由标题栏、页头、内容栏和页尾等几个 Section 构成，每一个 section 都是报表设计的一个逻辑单位，在快速报表中称之为 Band。

这些 Band，如果是页尾，则显示在每一页的底部，在快速报表设计中称之为 FooterBand；如果是标题栏，则主要显示于整个内容的开始处，且只显示于第一页的头部，设计中称为 TitleBand；如果是内容栏，它是报表的主要信息部位，由它与数据库中的信息进行连接，这在报表设计中称之为 DetailBand；如果是列头，即为内容栏的表头，设计中称为 ColumnHeaderBand。另外，还有其他构成，如页头、统计栏等。如果是页头，则显示在每一页的头部，其作用就如同 Word 中的页眉，设计中称为 PageHeaderBand；如果是统计，一般显示在整个报表的末尾，主要用于合计或者其他统计而使用的，在设计称为 SummaryBand。

报表的 band 主要用于将报表分成几个逻辑单元，除了这些 bands 之外，报表又是由放置在这些 band 之上的报表元素构成的，这些元素定义了在各 Band 上应该显示哪些内容。有两个比较重要的元素是标签（Lables）和数据字段（DetailFields）。标签主要用于显示一

些文本信息，这些信息都是静态信息，如标题，这些都是与数据库无关的；而数据字段，则是用于显示数据库中的信息，是动态变化的，它显示的内容来自于数据库。通常，标签一般都是数据字段的附属品，它使报表更为好看，并使数据字段在列数据时更易理解。

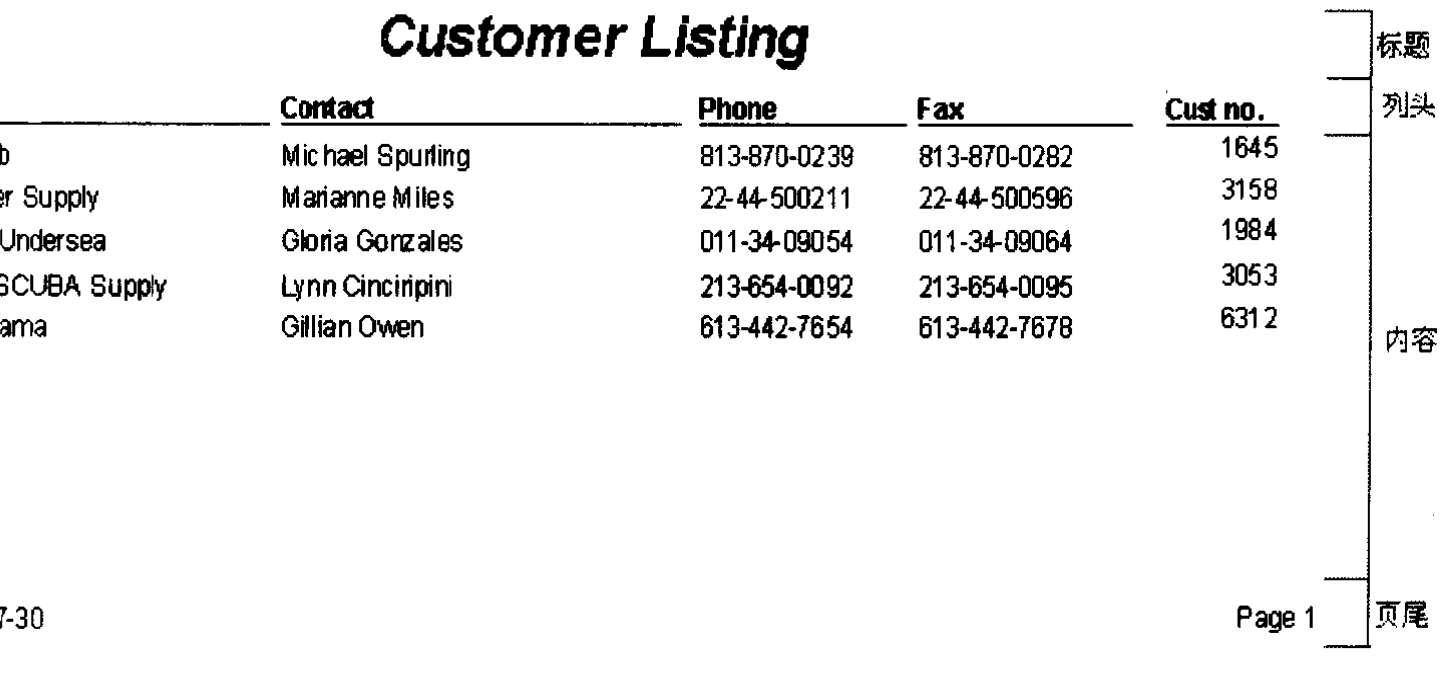


图 17-1　简单报表构成

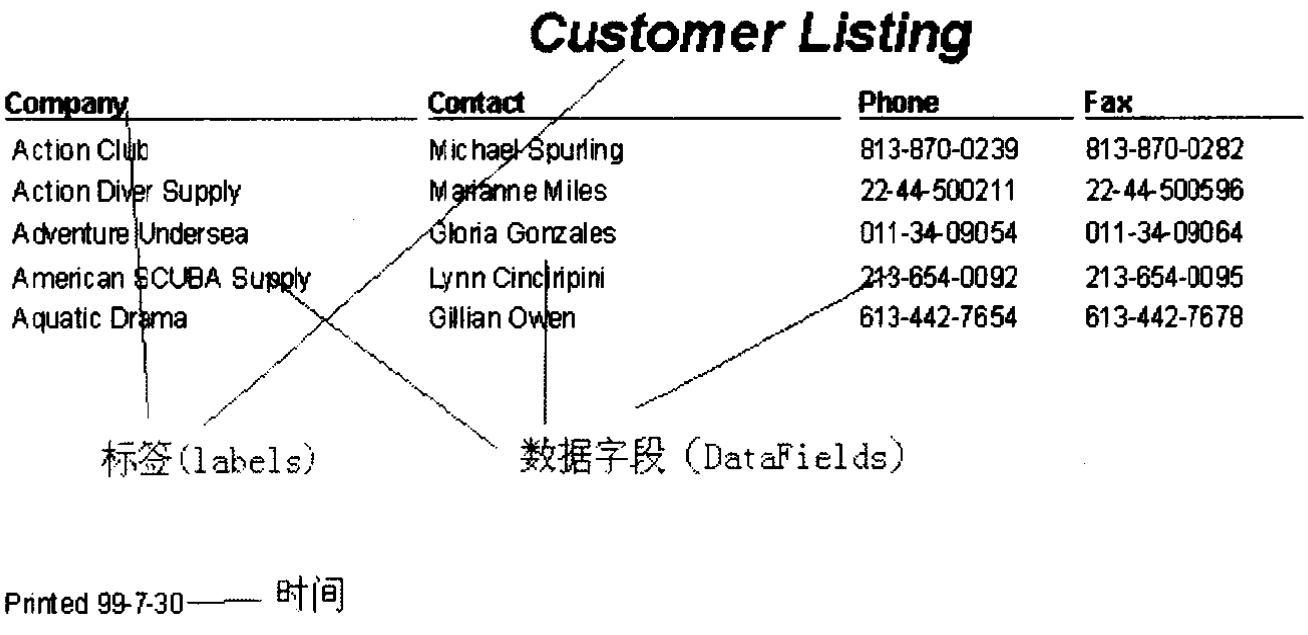


图 17-2　报表的形成元素

各 band 在 TquickRep 上是以真实的打印顺序出现的。然而一般的打印报表是以以下给定的顺序打印的这有助于理解它们为何以这样的顺序排列：

Page header repeated for all pages
|
Title （first page）
|
Column header repeated for all columns
|
Detail repeated for all records
|
Summary （last page）
|
Page footer repeated for all pages

2. 报表类型

按照报表的格式来分，报表的种类很多。现就常用的一些类型给以说明。

（1）简单报表。顾名思义，只是对数据库中数据进行简单的列表，并不进行进一步的处理。如图 17-1 为简单报表，一般由 TitleBand、ColumnBand 以及 DetailBand 构成。

（2）标签式报表。这种报表主要用来输出一些卡片式的信息，一般要求在一页上分成几列进行打印。这种报表仅包含一个 DetailBand。如图 17-3。

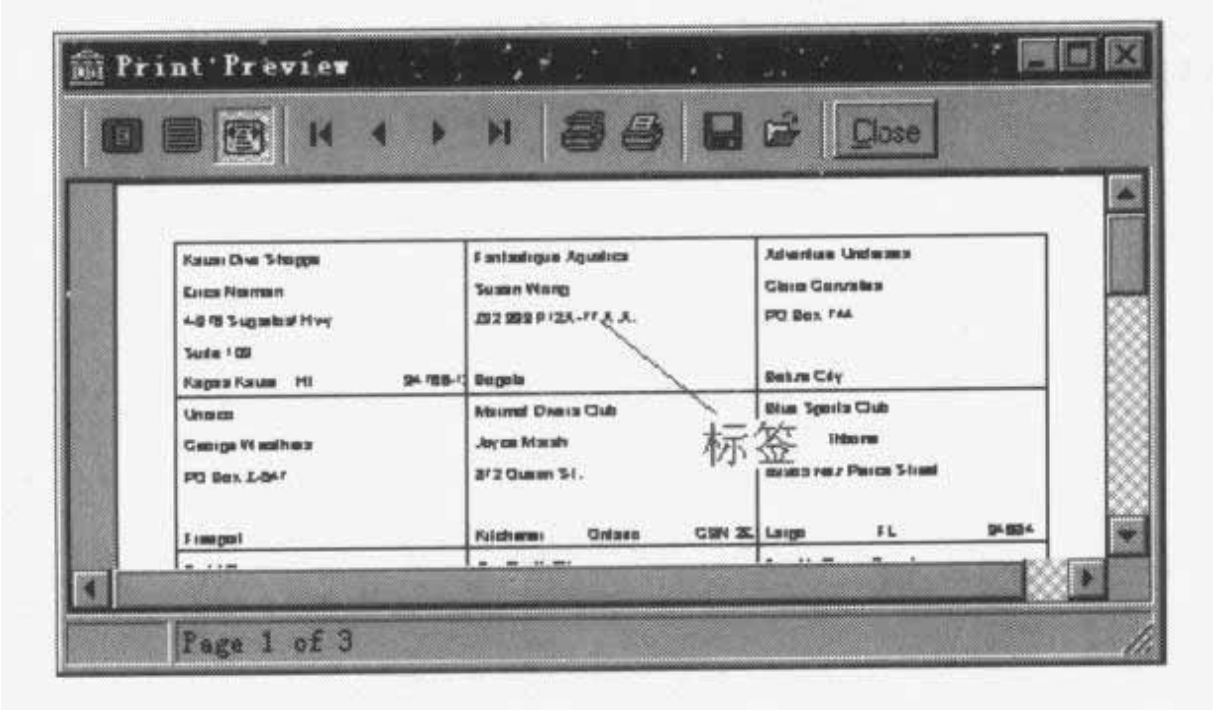


图 17-3　标签式报表

（3）主/明细报表。这种报表主要是用来输出一对多的关系。如，厂家与产品的关系，一个厂家可能生产多种产品，而销售商有可能与多个厂家的多个产品有联系，那么为了输出厂家的产品信息，应当分别为各厂家列出各自的产品信息。像这种厂家与产品的关系，即为主/明细关系。这种报表的组成一般比较复杂，主要由 TitleBand、ColumnBand、DetailBand、GroupHeaderBand 以及 SubDetailBand 等组成。如图 17-4 所示。

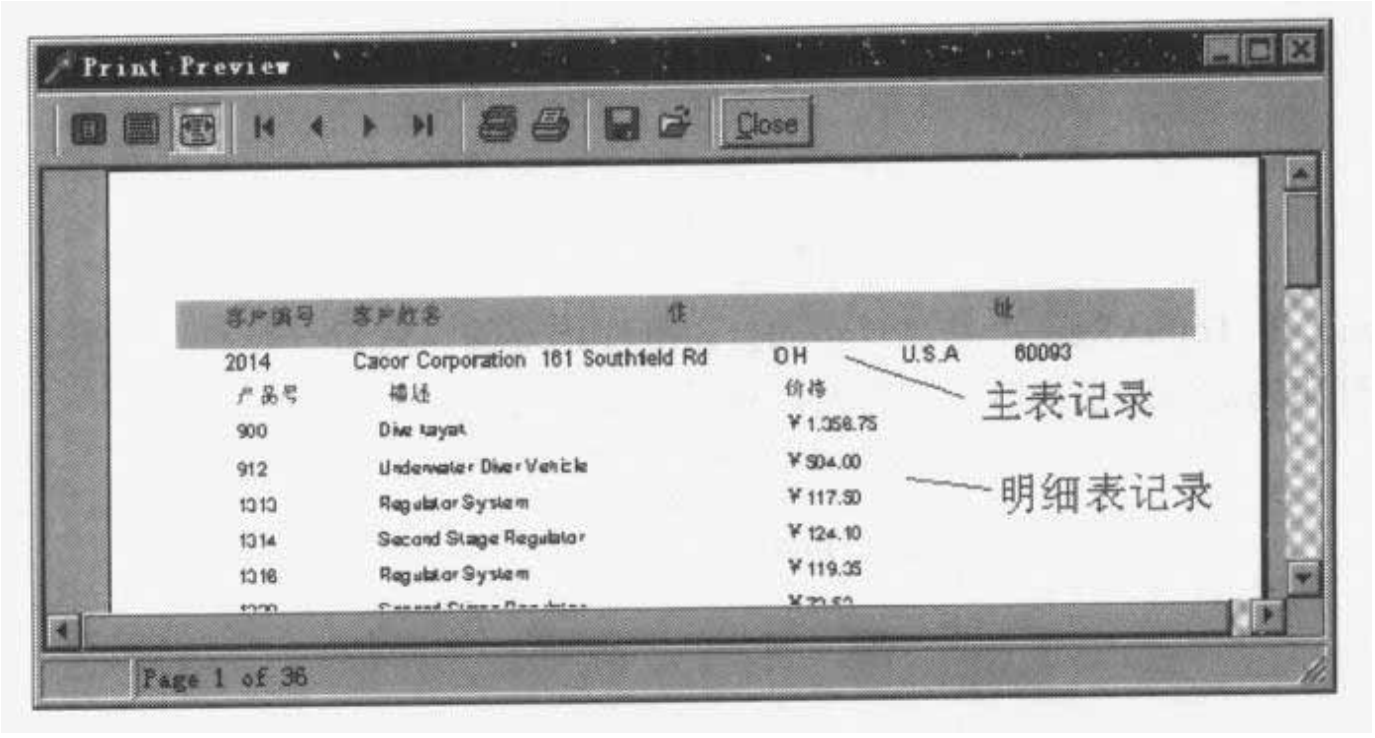


图 17-4　主/明细报表

（4）统计报表。这种报表除了将信息列举出来之外，还要对信息进行统计。因此此类

报表一般还应包括 SummaryBand，用来对数据进行统计处理。此类报表如图 17-5 所示。

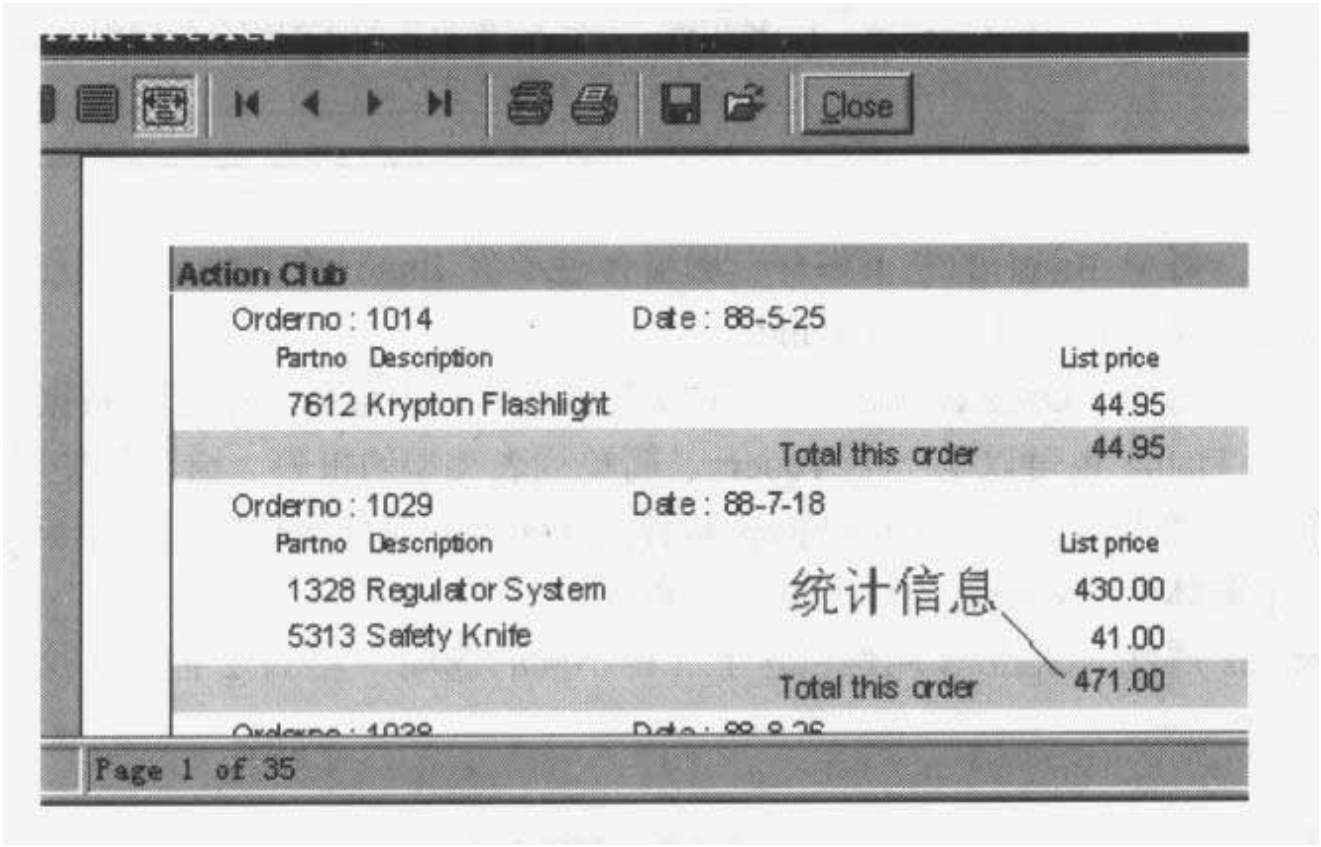


图 17-5　统计报表

3. 快速报表构件

快速报表都是由 QuickReport 的构件构成的，所以要创建一个快速报表应该掌握好报表元素的各个属性、方法和事件。下面简要介绍一下 QuickReport 构件包。

快速报表构件

Delphi4 所使用的 QuickReport 的版本一般为 3.x，在 Qreport3.0.4K 中总共有 23 个构件，它们在 Delphi4 IDE 构件面板的 Qreport 标签上可以找到。以下简单地阐述了各构件及其有用的属性和构件。

（1）TquickRep 构件。TquickRep 构件是快速报表包的心脏与灵魂，也是该包生成的所有报表的起点。将该构件拖至一个窗体上，然后设置该构件的属性，此时有几种方式，可以双击该控件，也可以选中该控件然后按鼠标右键并点击“report setting”菜单，也可通过属性页进行设置。该构件主要包括以下属性。

① Bands 属性。Bands 属性是具有如下值的属性集：hasColumnHeader、hasDetail、HasPageFooter、hasPageHeader、hasSummary 和 hastitle。表 17-1 描述了每种 Band 的类型。

表 17-1　　BandType 属性值

值	描述
Rbchild	连接到另一个 Band 上的子 Band；将该子 Band 挨着它所连接到的 Band 打印
Rbcolumnheader	在多列报表中，在每列的头上打印列标题（columnheader）的 Band
Rbdetail	在报表中为每个记录打印的明细 Band
RbGroupFooter	在每个组的末尾打印的组脚 Band（一个主/明细报表）
RbGroupHeader	在每个组的开始处打印的组头 Band（一个主/明细报表）
RbOverlay	一个覆盖 Band，从页的左上角开始打印，并覆盖页中所有其他的 Band
RbPageHeader	一个页头 Band，在每页的头部打印
RbPageFooter	一个页脚 Band，在每页的底部打印

续表

值	描述
RbSubDetail	一个子明细 Band，被用做一个主/明细报表的明细部分的明细 Band
RbSummary	一个总结 Band，仅在每个报表的尾部被打印一次
RbTitle	标题 Band，仅在每个报表的开始处打印一次

在设计时，每个 Band 的左下脚有一标签会显示该 Band 的类型，这样在向它们添加控件时，可以很容易区分它属于哪种 Band。

② Dataset 属性。DataSet 属性指明主报表将从哪一个数据集中获得数据值。这些数据集可以是一个 Ttable 也可以是一个 Tquery。简单列表类型的报表，通常只使用一个数据集，但对于一个主/明细报表来说，TquickRep 构件的 Dataset 属性应设置主数据集。至于数据集是位于哪一个窗体上或者数据模块上都无关紧要。

③ Options 属性。Options 属性包含了 3 种不同的选项。表 17-2 描述了这 3 种不同的选项。

表 17-2　　TQuickRep 的 Options 属性

属性值	描述
FirstpageHeader	表示第一页是否显示页头 band
LastPageFooter	表示最后一页是否显示页尾 band
Compression	表示是否以压缩方式存储报表。报表只占用程序的一部分空间，但要花费很长的时间来运行。这是尺寸和速度间的平衡

④ page 属性。page 属性主要用来设置报表的一些页面属性，如报表页的大小、页面距、打印方向以及栏的数目等等。

⑤ printersettings 属性。printersettings 属性主要用来设置打印的一些属性，如打印的份数、打印的范围等。

⑥ reporttitle 属性。reporttitle 属性由 Tqrsysdata 属性使用，在打印和预览的窗体标题上将显示该属性的内容。

⑦ showprogress 属性。showprogress 属性用于在准备和打印报表期间显示一个过程对话框。过程对话框含有一个 Cancel 按钮，允许用户停止生成报表。

⑧ OnNeedData 事件。OnNeedData 事件被用来从一个非 BDE 数据集中生成一个快速报表，如数据来自一个文本文件、数组或者字符串列等。当快速报表需要下一个“记录”的数据时，则触发该事件。若此时没有连接到一个 BDE 数据集，则需要自己填写报表中 TQRLabel 构件的文本值。OnNeedData 有一个名为 MoreData 的布尔参数，在报表完成时需要将其设为 false。注意，要使该事件有效的话，TQuickRep 的 DataSet 属性应该设置为空。如下例，为了将 SomeStringList 中的所有字符串打印出来，编写了以下代码：

```
var
    CurrentItem : Integer;
    SomeStringList : TStringList;

procedure TForm1.QuickRep1BeforePrint（Sender: TQuickRep;
```

```
var
    PrintReport: Boolean）;
begin
    CurrentItem := 0;
    PrintReport := SomeStringList.Count > 0;
end;
procedure TForm1.QuickRep1NeedData（Sender: TObject; var MoreData; Boolean）;
begin
    if CurrentItem <   SomeStringList.Count then
    QRLabel1.Caption := SomeStringList[CurrentItem];
    Inc（CurrentItem）;
    MoreData := CurrentItem <= SomeStringList.Count;
    end;
    procedure TForm1.Button1Click（Sender: TObject）;
    begin
        QuickRep1.Preview;
    end;
```

（2）TorSubDetail。TqrSubDetail 构件被用来在一个主/明细报表中显示明细记录。该构件的 3 种主要属性是：Bands、DataSet 和 Master。

① Bands 属性。Bands 属性是具有两个值的属性集—hasFooter 和 hasHeader，使用这两个值可以分别创建组脚（Group Footer）和组头（Group Header）Band。

② Dataset 属性。DataSet 属性用于指明本 Band 上的构件将从哪个 Dateset 上获取信息。应将其设置成主/明细关系中的明细数据集。

③ Master 属性。Master 属性表示指向主数据连接的 Band。它通常是一个 TquickRep 构件，如果创建了一个主/明细表，则该属性应指向第一个 TqrSubDetail 构件。

（3）QRBand。除了 TquickRep 构件之外，TQRBand 可能是快速报表中最重要的构件。每个报表都包含了一种或两种类型的 TQRBand 构件。在通过 TquickRep 构件的 Bands 属性向一个报表添加 Band 时，Delphi 实际上创建了一个 TQRBand。有些 Band 自动打印，有些在每页上打印，有些则在整个报表中只打印一次。

① BandType 属性。BandType 属性用来指明一个 TQRBand 构件是何种类型的 Band。Band 属性决定了在最后的报表中该 Band 出现的位置。表 17.1 中已经描述了 Band 的类型和在报表中出现的位置。

② ForceNewColumn 属性。ForceNewColumn 属性会使报表在打印该 Band 前创建一个新列。

③ ForceNewpage 属性。ForceNewPage 属性将使报表在打印该 Band 前创建一个新页。在一个简单的列举报表中，在明细 Band 中设置该属性为 True 时，则会使每个报表页显示一个记录。在主/明细报表中，将明细 Band 的属性设为 True 时将生成一个新页，且该报表的所有明细目录都出现在同一页中。

④ LinkBand 属性。当报表在打印一个 Band 时，它会检查当前的页，看上面是否有足够的空间打印这个 Band。LinkBand 属性被用来迫使报表链接 Band，并和当前的 Band 放置

在同一页上。如果当前页上没有足够的空间来打印当前的 Band 和链接 Band 时，报表将启动一个新页。可以将各 Band 链接到一起（如将 Band1 链接到 Band2，Band2 链接到 Band3...）。



在链接 Band 时，通常以向前的方式链接。换句话说，如果 Band1 在 Band2 之前打印，可以将 Band1 链接到 Band2。如果将 Band2 链接到 Band1，不会产生预期的效果，这是因为在计算 Band2 之前，Band1 已经被打印了。

⑤ BeforePrint 事件。在报表打印一个 Band 时会触发 BeforePrint 事件。一个名为 PrintBand 的布尔参数将传递给该事件，它用来决定是否要打印当前这个 Band。PrintBand 的缺省值为 True，若要禁止打印该 Band，应将其设为 False。在主/明细报表中经常使用该事件，当一个给定的主记录没有明细记录时，可能会想禁止对明细和子明细 Band 的打印。

例如，在\Borland\Delphi4\Demos\Quickrpt\QrDemo.dpr 中的 MD.pas 单元文件中，为了不将公司 Uinisco 的信息显示出来，则可以为 QRSubDetail1 事件加入以下代码：

```
procedure TMDForm.QRSubDetail1BeforePrint（Sender: TQRCustomBand;
  var PrintBand: Boolean）;
begin
    if uppercase（'Uinisco'）=uppercase（customer.fieldbyname（'company'）.asstring）
then
      printband:=false
    else
      printband:=true;
end;
```



在用户设计时，如果用户右击报表构件，并从弹出菜单中选择“preview”，那么输入到 Band 的 BeforePrint 事件中的代码将会被调用。只有在运行时调用报表构件的 preview 方法时才会调用 BeforPrint 事件，设计阶段预览时，可能会看到 Band，在运行阶段，实际预览和打印报表时，可能不会看到 Band。

（4）TQRChildBand。TQRChileBand 构件被用来创建与另一个 Band 相联系的 Band，这与一个明细和子明细 Band 间的关系很类似，但没有主明细关系间的形式。通过拖放一个 TQRChildBand 放到一个报表上，或是设置一个 Band 的 haschild 属性为 True，都可以创建一个子 Band。

要在一个明细 Band 内显示备注信息时，通常需要使用子 Band 而该备注信息后往往又跟随有其他记录信息。若设置 QRDBText 构件的 AutoStretch 属性为 True，则它会扩展其自身而越过其他控件的顶部。解决的办法是在明细 Band 内显示备注信息，而将其他信息放在与该明细 Band 相连的子 Band 内。这就允许按需要扩展备注信息，而不需要重写跟在其后的信息。

ParentBand 属性用来将子 Band 链接到它的父 Band 上。该子 Band 可链接到任何类型的报表 Band 上，甚至是其他的子 Band 都可以。

（5）TQRGroup 构件。TQRGroup 构件用来在一个报表内创建组。组主要用于主明细报表内，也可以在一个简单的列表类型报表内使用。

① Expression 属性。Expression 属性用来创建一个表达式指明何时需要打破组，当 Expression 属性发生变化时即会打破组。例如，对于一个简单的列表类型报表，通过在 Expression 属性中输入如下的表达式，可以使得当一个特定字段的第一个字母改变时便打破组：

copy（fieldname,1,1）

② Master 属性。Master 属性用来指明 TQRGroup 构件的数据源。该属性通常被放到它所拖放到的 TquickRep 构件上。

（6）TQRLabel 构件。TQRLabel 构件用于报表内显示的静态文本，就如同使用 Tlabel 构件在一个窗体上显示静态文本值一样。这里所说的静态文本值，并不是说该值的内容不能改变，而是说这个文本值不是经常地被改变，或者不是来自数据库。

（7）TQRDBText 构件。TQRDBText 构件是一个指明数据的控件，它用于显示一个字符串字段、数值字段、日期字段和备注字段的值。像其他指明数据的控件一样，该字段通过将 Dataset 和 DataField 属性相应钩到一个数据集和一个数据字段上来实现。

AutoStretch 属性。AutoStretch 是一个布尔属性，用于确定 TQRDBText 构件是否要进行垂直扩展，以适应一个备注字段的完整内容。若 AutoStretch 属性为 True，控件会显示完备的备注，若 TQRDBText 构件的大小不够。会打破页的限制来显示备注。



设置 AutoStretch 属性会引起 TQRDBText 构件在垂直方向上重新调整其自身的大小，甚至越过其他快速报表构件的顶部。在特定的情况下，需要在同一个明细 Band 内显示带有其他记录信息的备注信息，此时应该在明细 Band 内显示备注信息，而在子 Band 内显示其他信息。

（8）TQRExpr 构件。TQRExpr 构件用于为报表创建简单或复杂的计算。

① Expression 属性。对于整个构件来说，Expression 属性是一个关键的属性。当单击 Expression 属性时，可以看到一个省略号按钮（...），在其上单击鼠标，便可以激活 Expression Builder 属性编辑器（如图 17-6 所示），这与 2.0 版相比在风格、功能和智能化上强化了不少。Expression Builder 可在创建报表所需要的表达式中作为向导。

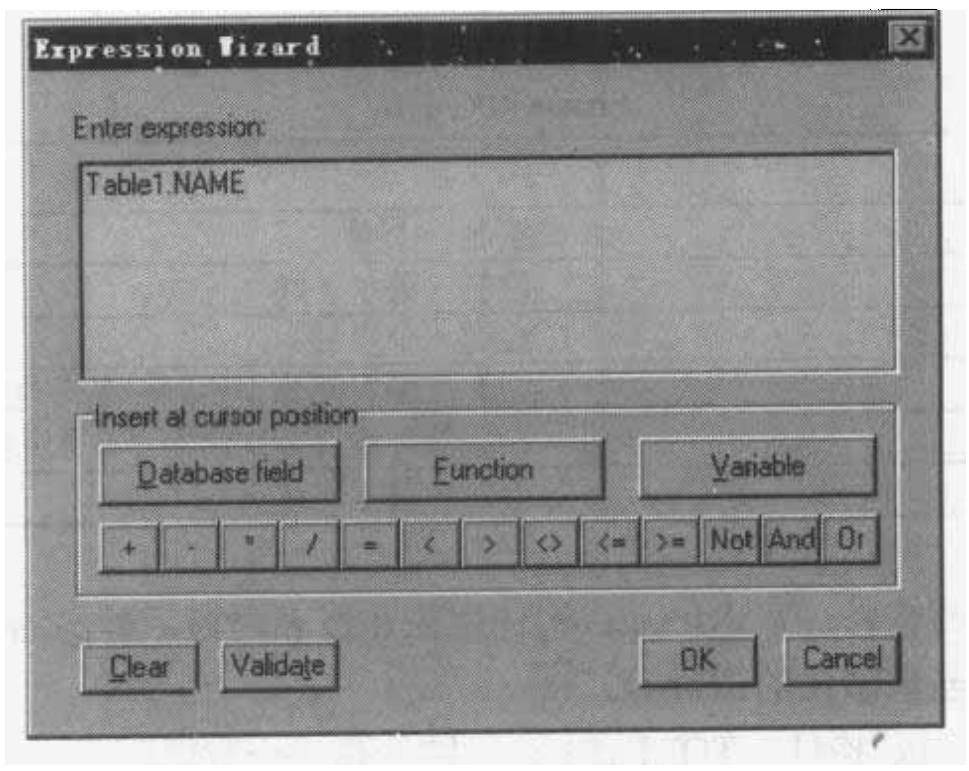


图 17-6　Expression Builder

② ResetAfterPrint 属性。ResetAfterPrint 是一个布尔属性，它用于确定在每次打印完

TQRExpr 构件值后，是否应该将它的值重置回缺省的值。若正在创建的是一个主/明细类型的报表，且该报表里 TQRExpr 构件正在创建某种运行时的总结，一般在这种情况下会希望重置值。

（9）TQRSysData 构件。在报表被打印时，TQRSysData 会用来显示有关报表以及操作系统的一些信息。

Data 属性。Data 属性用于指明 TQRSysData 构件显示的信息的类型。表 17-3 列出了每种信息类型相应的 7 种不同的属性值和描述。通常是在报表的 Header、Footer 和标题内使用该构件。

表 17-3　　Data 属性

值	描述
QrsDate	当前时间
QrsDataTime	当前日期
QrsDetailCount	报表内的记录数
QrsDetailNo	当前记录号
QrsPageNumber	当前页号
QrsReportTitle	显示 Tquickrep 构件的 ReportTitle 属性值
QrsTime	当前时间

（10）TQRMemo 构件。TQRMemo 构件用来在报表内显示备注信息，这个构件的行为非常类似于一个普通的 Tmemo 控件，可通过 Lines 属性自行添加信息。它也是一个静态信息。

（11）TQRDBRichText 构件。TQRDBRichText 用来在报表内显示 RTF 文件。TQRRichEdit 有一个名为 parentRichEdit 的属性。它允许将 TQRRichEdit 钩到另一个 TrichEdit 构件上。从而使 TQRRichEdit 可以反映另一个控件的 RTF 备注信息。

（12）QRDBRichText 构件。这是 TQRRichText 的数据明了（data-aware）版。

（13）TQRShape 构件。TQRShape 构件用来在报表内绘制一些简单的图形。绘制的图形依赖于 Shape 属性。表 17-4 列出了 Shape 属性的值。

表 17-4　　Shape 属性的值

值	描述
QrsCircle	绘制一个圆
QrsHorLine	绘制一条水平线
QrsRectangle	绘制一个矩形
QrsRightandleft	在左部和右部分别绘制一条垂直线
QrsTopandBottom	在顶部和底部分别绘制一条水平线
qrsVertLine	绘制一条垂直线

（14）TQRImage 构件。TQRImage 构件用来在报表内显示静态图像信息，可以使用这个构件向报表添加一些数字化的徽标等。

（15）TQRDBImage 构件。TQRDBImage 构件是一个数据明了（data-aware）的构件，它用来从数据库向报表中添加图像。它的使用方式类似于一个普通的 TDBImage。

（16）TQRCompositeReport 构件。TQRCompositeReport 构件可以将多个报表合并成一

个报表。使用这个构件，可以创建一个包含许多不同条件的大报表，而不需通过编程来实现打印。可以简单的将条件报表当作一个分离的报表进行创建，然后只需把想打印到总包报表内的部分添加进去。通过 OnAddReports 事件将报表添加到 TQRCompositeReport 内。在 OnAddReports 事件内，按如下代码向 Reports 属性添加报表：

```
procedure TMainForm.QRCompositeReport1AddReports（Sender: TObject）;
begin
  with QRCompositeReport1 do
  begin
    // TQRCompositeReport .Reports.Add（QuickRep:TQuickRep）
    Reports.Add（QuickRep1）;
    Reports.Add（QuickRep2）;
    Reports.Add（QuickRep3）;
    Reports.Add（QuickRep4）;
  end;
end;
```

（17）TQRPreview 构件。TQRPreview 构件用于创建自定义的预览屏幕来代替快速报表缺省的预览屏幕。有关创建自定义预览报表的更多的信息，请参见后面章节。

（18）TQRChart 构件。TQRChart 构件是一个快速报表定制的 Tchart 构件，它用于向报表嵌入图表。在 TQRGroup Band 内，该构件可以工作得很好，因为在这个 Band 内可以显示后面明细记录图形化的总结信息。



跟我来

创建报表的步骤非常简单，一般需要通过三个步骤。

（1）将一个 TquickRep 构件拖到屏幕上；

（2）创建报表所需的 Bands；

（3）将快速报表的控件拖放到各自的 Band 上，并连接相应的数据库以显示信息。

另外，Delphi 提供了创建报表的两种简单方式，即“QuickReport Wizard”和“Report Templete”。

1. “QuickReort Wizard”

“QuickReort Wizard”位于“Object Repository”的“Business”页。它是创建简单类型报表的快速而又简单的方式。图 17-7 显示了“QuickReport Wizard”在“Object Repositoty”中的外观。

通过使用“QuickReport Wizard”和“DBDemos”中的 Customer.db 表，可以创建一个简单的报表，步骤如下：

（1）新建一个应用程序；

（2）在 IDE 的主菜单内选择“FilelNew”，并打开“Object Repository”；

（3）选择“Business”选项卡，然后单击“QuickReport Wizard”；

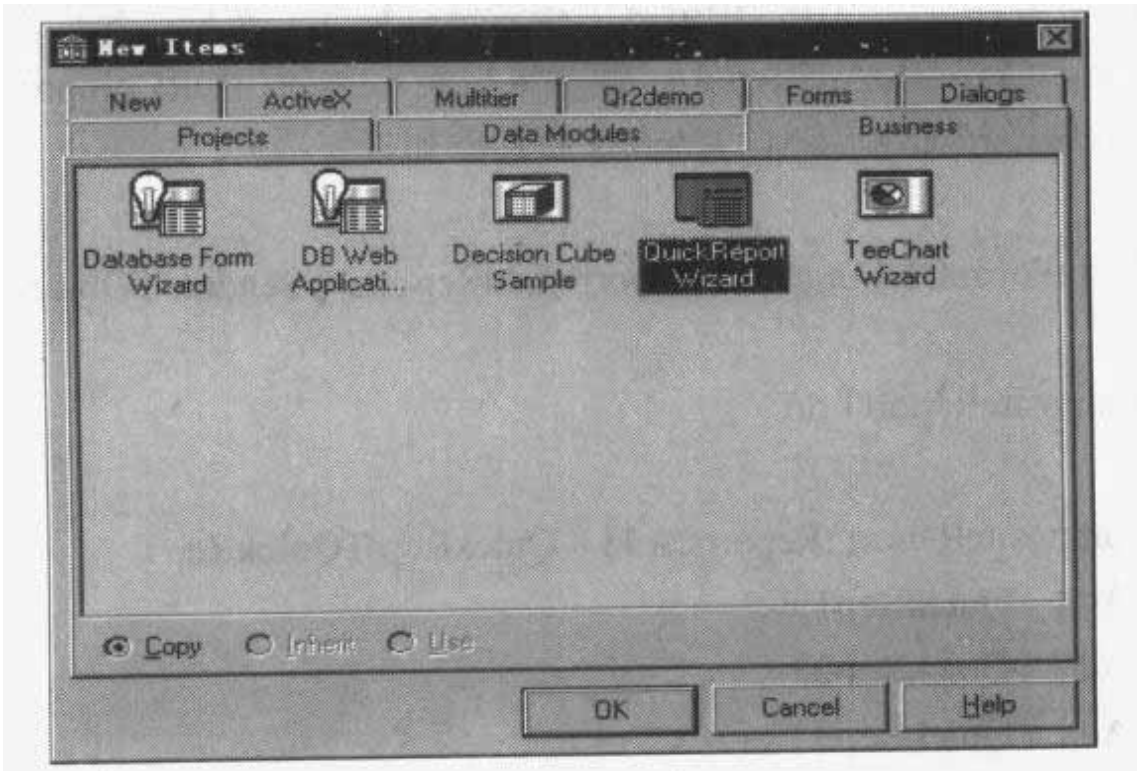


图 17-7 “Object Repository”中的“Bussiness”页

（4）选择数据库别名 DBDemos 和表格 Customer.db，然后单击“Next”按钮；

（5）选择字段，并单击“Finish”按钮，然后回到 Delphi IDE 中；

（6）右击报表窗体，然后从弹出菜单中选择“preview”，便可以看到如图 17-8 所示的报表。

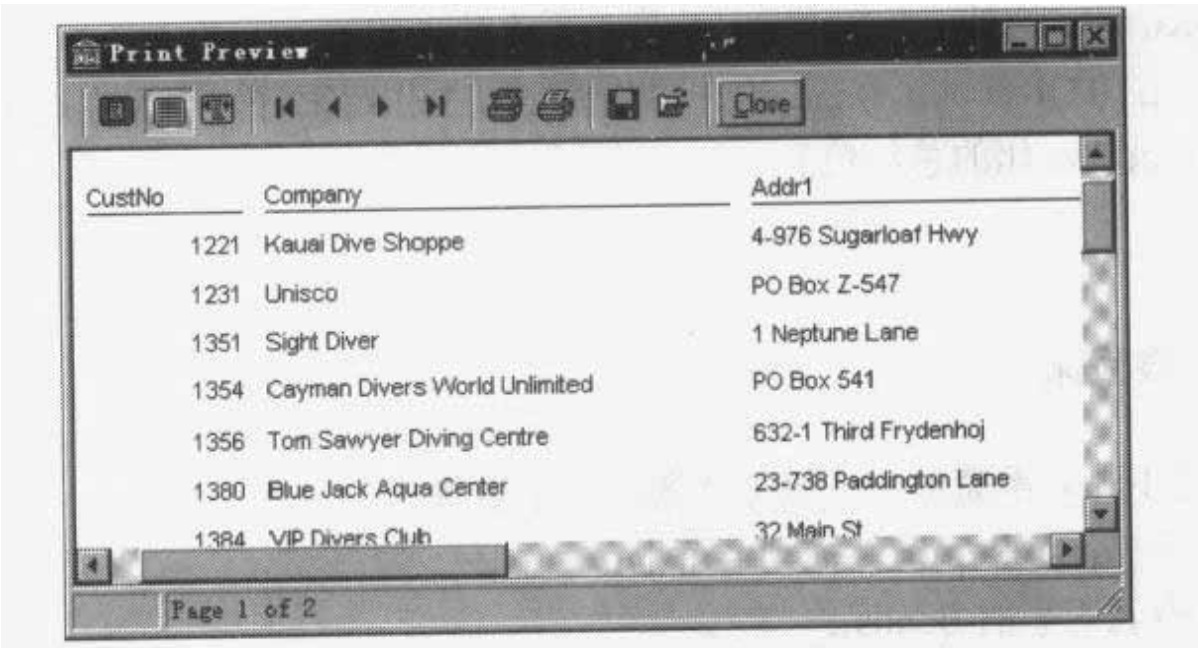


图 17-8 “QuickReport Wizard”生成的 customer 报表

2. “Report Templates”

类似于“QuickReport Wizard”，“Report Templates”也位于“Object repository”内。选项卡 Forms 包含了 3 种快速报表模板。图显示了“Report Templates”在“Object Repository”内的外观。使用“Report Templates”比使用 Wizard 需要多做些工作，但是这样做提供了更大的灵活性和控制。本节介绍怎样使用 Labels 和 Master/Detail 模板来生成一个报表。

（1）快速报表 List。List 模板生成一个列表类型的报表，它与使用“Report Wizard”生成的报表相同，只是缩小了每个字段的自动生成的文本控制。大多数情况下，使用“Report Wizard”生成这种类型的报表更快捷和简单。

（2）快速报表 Labels。Labels 模板创建一个打印标签的 QuickReport 报表。这个报表

内仅包含一个明细 Band，该 Band 的宽度和高度被设置成标准的三列标签窗体中单个标签的尺寸。要得到不同尺寸的标签，需调整明细 Band 的宽度和高度属性，以及每页的列数。调整 TquickRep 构件的 page 属性可以设置页面尺寸。

作为使用该模板的示例，我们基于 DBDemos 的 Customer 表格创建了一个标签报表。Customer 表格中包含了一些国际的公司的信息。按如下步骤创建的表格包含公司的名称、地址、城市、Zip 码和数据库中每个记录的联系人名的信息。若有联系人，则将为此人制作一个标签，如下所示：

① 在 Delphi 内创建一个新的应用程序；

② 把两个 Tbutton 拖到窗体上，并设置标题分别为 Preview 和 Print；

③ 从 Delphi 的主菜单内选择“FilelNew”来打开“Object Repository”。选择选项卡 Forms，然后选择窗体 QuickReport Labels；

④ 在快速报表的窗体上，设置 MasterTable 构件的 DatabaseName 属性为 DBDemos，TableName 属性为 Customer.db 以及 Active 属性为 True；

⑤ 把 7 个 TQRDBText 构件拖放至 Detail Band 上并排放置。设置 DataSet 属性为 masterTable，并设置它们的 datafield 属性；

⑥ 为了只打印美国的客户记录，在 Detail Band 的 BeforePrint 事件内添加如下的代码：

```
PrintBand:=MasterTable['Country']='US';
```

⑦ 在“Preview”按钮的 OnClick 事件内加入下列代码：

```
QRListForm.Report.Preview;
```

⑧ 在“Print”按钮的 OnClick 事件加入下列代码：

```
QRListForm.Report.Print;
```

⑨ 在 Unit1 的 Uses 部分中加入 Unit2。

运行程序并单击“Preview”按钮，便可以看到类似于图 17-3 所示的结果。

（3）快速报表 Master/Detail。Master/Detail 报表模板用于创建 QuickReport 报表，其中包含有标题、列、明细、子明细和页脚 Band。缺省的情况下报表窗体上有两个 Ttable 构件名为 MasterTable 和 Detail Table，Band 正是链接在这些表格上。

作为使用这些模板的示例，基于 DBDemos 的 Vendors 和 parts 表格创建了一个报表。Vender 表格包含了一列贩卖商唯一的标识符，Parts 表格含有一列唯一标识的部件号以及它们相应的贩卖商标识符。按下面的步骤生成的报表包含了一列贩卖商以及它们的产品。若在 Vendors 表中的一个贩卖商在 parts 表格中没有包含任何项，则在报表中不包含此贩卖商。步骤如下：

① 在 Delphi 内创建一个新的应用程序；

② 把两个 Tbutton 拖到主窗体上，并设置标题分别为 Preview 和 Print；

③ 从 Delphi 的主菜单内选择“FilelNew”来打开“Object Repository”。选择选项卡 Forms，然后选择窗体 QuickReport Master/Detail；

④ 在快速报表的窗体上，设置 MasterTable 和 DetailTable 构件的 DatabaseName 属性为 DBDemos，它们的 TableName 属性分别为 Vendors.db 和 Parts.db，以及 Active 属性为 True；

⑤ 在快速报表的窗体上，设置 TQuickRep 构件的 Dataset 属性为 MasterTable，设置 SubDetailBand 构件的 DataSet 属性为 DetailTable，设置 Bands 构件的 HasHeader 属性为 True；

⑥ 在 ColumnHeaderBand 上，添加 3 个 TQRLabel 构件并排放置并设置它们的 Caption

属性；

⑦ 在 Detail Band 上，拖放 7 个 TQRDBText 构件并排放置，并设置 DataSet 属性都为 masterTable，以及它们的 datafield 属性。设置 DetailBand 的 ForceNewPage 属性为 True，这将使得在每个报表页上只显示一个贩卖商；

⑧ 在 GroupHeaderBand 上，添加 3 个 TQRLabel 构件并排放置并设置它们的 Caption 属性；

⑨ 在 SubDetailBand 上，为 PartNo、Desctiption 和 Cost　3 个字段添加 3 个 TQRDBText 构件并排放置。设置 Dataset 为 DetailTable，并设置 DataField 属性；

⑩ 为了不打印没有关系的贩卖商，在 DetailBand 的 BeforePrint 事件中添加如下的代码：

```
PrintBand:=DetailTable.Recordcount>0;
```

⑪ 在“Preview”按钮的 OnClick 事件内加入下列代码：

```
QRListForm.Report.Preview;
```

⑫ 在“Print”按钮的 OnClick 事件内加入下列代码：

```
QRListForm.Report.Print;
```

⑬ 在 Unit1 的 Uses 部分中加入 Unit2。

运行程序并单击“Preview”按钮，便可以看到类似于图 17-4 的结果。



创建自定义预览



遇到难题

我们知道，在进行报表的预览时，整个窗体的操作界面都是英文状态或英文提示的，这无疑对所编写的全中文应用软件来说是一大遗憾。所以，我想您肯定希望要么能得到一个全中文提示的快速报表控件，要么能获取提供自己编写预览窗体的接口。但对于前者 Qusoft 公司并未提供，幸运的是，它提供了后者。因此，要实现中文的预览窗体是完全可以的。那么如何将自己定义的预览窗体加入到应用程序中呢？



钥匙在此

在 Delphi 中通过使用 TQRPreview 构件很容易创建自定义预览。创建一个自定义预览需按如下步骤进行：

（1）创建一个新的窗体，并将 TQRPreview 构件拖放在其上；

（2）在窗体中添加按钮和菜单项，用于实现报表的导引、打印和显示；

（3）在所有需要使用自定义预览的 OnPreview 事件句柄中添加如下代码：

PrevForm.QRPreview1.QRPrinter:=TQRPrinter（Sender）；

其中 PrevForm 是自定义预览窗体的名字，而 QRPreview1 是 TQRPreview 构件的名字。步骤（3）是使自定义预览得以工作的关键步骤。它把自定义预览的 Prenter 对象赋给报表的 Printer 对象。



跟我来

下面看一个实例，看预览窗体是如何实现的。

步骤如下：

（1）新建一个窗体，并设置 Name 为 MyPreview；

（2）在窗体上添加一 Toolbar 控件，模仿缺省预览窗口创建相应的 Toolbutton，并设置各按钮的 Hint 提示；

（3）添加一 Panel 控件，并将对齐方式设置为 alBottom。再在此 Panel 上放一 ProgressBar（左对齐）和 Panel（右对齐），分别显示报表装载进度和其他提示信息；

（4）添加 QRPreview 控件，对齐方式设置为 alClient；

（5）添加 OpenDialog，设置 Filter 属性为*.QR；添加 SaveDialog，设置 Filter 属性为*.QR|*.TXT|*.HTM|*.CSV，设置 DefaultExt 属性为*.QR；

（6）双击各个 Toolbutton，然后输入相应代码。

可以按以下方法调用自定义预览窗口。

重载 TQuickRep 的 OnPreview 事件，输入如下代码：

```
procedureTRptForm.RptFormPreview（Sender:TObject）；
begin
    withTMyPreview.Create（Application）do
    begin
        QRPreview1.QRPrinter:=TQRPrinter（Sender）；
        CurRep:=self;
        Show;
    end;
end;
```

附各成员方法的具体实现：

```
unitMyprv;

interface

uses
Windows,Messages,SysUtils,Classes,Graphics,Controls,Forms,Dialogs,ComCtrls,ToolWin,qr
prntr,Quickrpt,StdCtrls,ExtCtrls,qrextra,qrhtml;
```

```
Type

TMyPreview=class（TForm）
    QRPreview1:TQRPreview;
    ToolBar1:TToolBar;
    //其余成员变量和成员方法的声明略
private
    {Privatedeclarations}
    FPageCount:integer;//生成报表的总页数
public
    {Publicdeclarations}
    CurRep:TQuickRep;//所预览的报表
    procedureUpdatePanelShow;
end;

implementation

{$R*.DFM}
procedureTMyPreview.UpdatePanelShow;
begin//更新显示
    Panel2.Caption:=
        '第'+inttostr（QRPreview1.PageNumber）+
        '页总'+inttostr（FPageCount）+'页';
end;

procedureTMyPreview.ToolButton2Click（Sender:TObject）;
begin
    QRPreview1.ZoomToFit; //ZoomToFit  放大至全屏
end;

procedureTMyPreview.ToolButton3Click（Sender:TObject）;
begin
    QRPreview1.Zoom:=100; //ZoomTo100%    缩放至实际大小
end;

procedureTMyPreview.ToolButton4Click（Sender:TObject）;
begin//ZoomToWidth    缩放至页宽
    QRPreview1.ZoomToWidth;
end;
```

```
procedureTMyPreview.ToolButton6Click（Sender:TObject）;
begin//Firstpage
    QRPreview1.PageNumber:=1;
    UpdatePanelShow;
end;

procedureTMyPreview.ToolButton8Click（Sender:TObject）;
begin//priorpage
    QRPreview1.PageNumber:=QRPreview1.PageNumber+1;
    UpdatePanelShow;
end;

procedureTMyPreview.ToolButton7Click（Sender:TObject）;
begin//nextpage
    QRPreview1.PageNumber:=QRPreview1.PageNumber-1;
    ifQRPreview1.PageNumber=0then
    QRPreview1.PageNumber:=1;
    UpdatePanelShow;
end;

procedureTMyPreview.ToolButton9Click（Sender:TObject）;
begin//Lastpage
    QRPreview1.PageNumber:=FPageCount;
    UpdatePanelShow;
end;

procedureTMyPreview.QRPreview1PageAvailable（Sender:TObject;
PageNum:Integer）;
begin//getpagecount
    FPageCount:=PageNum;
    UpdatePanelShow;
end;

procedureTMyPreview.FormClose
（Sender:TObject;varAction:TCloseAction）;
begin
    CurRep:=Nil;
    Action:=caFree;
end;
```

```
procedureTMyPreview.ToolButton14Click（Sender:TObject）;
begin//closethewindow
    Close;
end;

procedureTMyPreview.ToolButton11Click（Sender:TObject）;
begin//printsetup
    QRPreview1.QRPrinter.PrintSetup;
end;

procedureTMyPreview.ToolButton13Click（Sender:TObject）;
begin//print
    QRPreview1.QRPrinter.Print;
end;

procedureTMyPreview.ToolButton16Click（Sender:TObject）;
begin//savebutton
    ifSaveDlg1.Executethen
    begin
        if（SaveDlg1.FIlterIndex<>1）and（CurRep=Nil）thenExit;
        caseSaveDlg1.FilterINdexof
        //--*.QR
        1:QRPreview1.QRPrinter.Save（SaveDlg1.FileName）;
        //--*.TXT
        2:CurRep.ExportToFilter
         （TQRAsciiExportFilter.Create（SaveDlg1.FileName））;
        //--*.HTM
        3:CurRep.ExportToFilter
         （TQRHTMLExportFilter.Create（SaveDlg1.FileName））;
        //--*.CSV
        4:CurRep.ExportToFilter
         （TQRCSVExportFilter.Create（SaveDlg1.FileName））;
        end;
    end;
end;

procedureTMyPreview.ToolButton17Click（Sender:TObject）;
begin//loadbutton
    if OpenDlg1.Execute then
```

```
    QRPreview1.QRPrinter.Load（OpenDlg1.FileName）;
end;

procedureTMyPreview.QRPreview1.ProgressUpdate（Sender:TObject;
Progress:Integer）;
begin//updageprogressbar
    ProgressBar1.Position:=Progress;
end;

procedureTMyPreview.SaveDlg1TypeChange（Sender:TObject）;
begin//setDefaultExtpropertyofSavedialog
    withSaveDlg1do
    caseFilterIndexof
    0:DefaultExt:='.QR';
    1:DefaultExt:='.TXT';
    2:DefaultExt:='.HTM';
    3:DefaultExt:='.CSV';
    end;
end;

end.//endofUnit
```

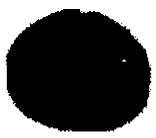
在创建自定义预览窗体中关键的语句是

```
PrevForm.QRPreview1.QRPrinter:=TQRPrinter（Sender）;
```

预览窗体中的其他操作其实在 QRPreview 控件中都提供了相应的方法和属性。



进行 QuickReport 附件的开发



遇到难题

如果经常去 QSD 公司的网站的程序员可以下载一些在 QuickReport 上使用的扩展的控件，如画斜线的控件、检查框控件以及可旋转字体的标签等等，这些会大大地丰富报表界面。但这些控件又并非是 QSD 公司推出的，因此，QSD 在这个 QuickReport 控件中留了增加控件的接口。所以，你同样可以在 QuickReport 上开发一些报表元素控件。



钥匙在此

可以通过一些附件来扩展 QuickReport。这些附件可以从 QSD 公司以及其他 Delphi 第三方开发公司获得。另外，如果自己找不到适合自己需要的话，可以为 QuickReport 开发一些附件。本节主要讨论如何编写四个不同类型的附件。

1. 可打印组件

虽然 QuickReport 已经提供了强大的报表设计功能，但是在实际应用中往往需要扩展一些功能，如需要 QuickReport 的报表能够输出条码、高级图形文件格式、特殊文本格式以及异样的标签等等。

一个可被 QuickReport 打印的控件，可以从最基本的可打印的控件 TQRPrintable 中继承下来，也可直接继承或者封装现有的组件。为了编写可打印控件，应该包括以下全部或者部分步骤：

（1）创建一个功能性可打印的组件；

（2）使该组件能出现在最终用户的报表设计中；

（3）包括输出功能。

2. 输出筛选

除了让 QuickReport 能输出 ASCII 文本文件之外，可能还需要将报表以其他形式，如 Word、Execel、RTF 等格式输出报表。Quickreport 已经包括了一些钩子函数，以及一些为任一文件格式产生输出筛选的基础类。

3. 函数库

QuickReport 表达式求值程序可以让你将函数库进行扩展。像数学函数、底层访问文件、字符串处理、时间计算等都可以加入。QuickReport 已经包括了可以轻易写函数库的类，一旦这些函数已经完成，你就可以在 QuickReport 的表达式中使用这些函数。

4. 报表编辑器

最终用户使用的 QuickReport 的报表编辑器可以很容易的被其他报表编辑器所替代。基本上 QuickReport 在最终用户的报表编辑时使用了三层接口。通过包含非可视化的 TQRDesigner 类，可以加入自己的可视化编辑器，而无需进行低层编程、使用 API 以及处理 Windows 消息等操作。



跟我来

下面通过两个例子看是如何扩展 QuickReport 的功能的。

1. 编写可打印组件

最有用的 QuickReport 附件是能够创建可打印组件的附件。使用面向对象的设计方法，QuickReport 会混淆报表打印概念。QuickReport 不知道如何打印一些特殊的东西。而 TQRPrintable 基类知道打印的内容，真正被打印的内容必须是 TQRPrintable 的子类，并且知

道如何打印它自己。

在 QuickReport 中所有的可打印组件都是用这种方式设计的。一个可打印组件继承了基类的属性并加入了自己的属性。

（1）一个例子—CheckBox 组件。这个例子就是为 QuickReport 编写一个 CheckBox 组件，它有个 Checked 属性，而且该组件应该重载（override）paint 方法（为了设计时的显示）和 print 方法(为了打印和预览)。并在 paint 方法和 print 方法中画出 checkbox 和取决于 checked 属性值的 check 标记。以下为该组件的源程序。

```
unit qrcb;
interface
uses classes, quickrpt,Graphics;
type
  TQRCheckBox = class（TQRPrintable）        //基类为 TQRPrintable
  private
    FChecked : boolean;
    procedure SetChecked（Value : boolean）;
  public
    constructor Create（AOwner : TComponent）; override;
    procedure Paint; override;
    procedure Print（OfsX, OfsY : integer）; override;
  published
    property Checked : boolean read FChecked write SetChecked;  //增加了 checked 属性
  end;
procedure Register;
implementation
constructor TQRCheckBox.Create（AOwner : TComponent）;
begin
  inherited Create（AOwner）;
  FChecked := False;
end;
procedure TQRCheckBox.SetChecked（Value : boolean）;
begin
  if Value <> FChecked then
  begin
    FChecked := Value;
    Invalidate;
  end;
end;
procedure TQRCheckBox.Paint;
begin
  //在 canvas 上画出 checkbox 的外观
```

```
with Canvas do
begin
  Pen.Color := clBlack;
  Pen.Width := 1;
  Rectangle（0, 0, Width - 1, Height - 1）;
  if Checked then
  begin
    MoveTo（0, 0）;
    LineTo（Width - 1, Height - 1）;
    MoveTo（0, Height - 1）;
    LineTo（Width - 1, 0）;
  end;
end;
//在 ParentReport.QRPrinter.canvas 画出 checkbox
procedure TQRCheckBox.Print（OfsX, OfsY : integer）;
var
  CalcLeft,
  CalcTop,
  CalcRight,
  CalcBottom : Longint;
begin
  with ParentReport.QRPrinter do
  begin
    Canvas.Pen.Color := clBlack;
    Canvas.Pen.Width := 1;
    CalcLeft := XPos（OfsX + Size.Left）;
    CalcTop := YPos（OfsY + Size.Top）;
    CalcRight := XPos（OfsX + Size.Left + Size.Width）;
    CalcBottom := YPos（OfsY + Size.Top + Size.Height）;
    Canvas.Rectangle（calcLeft, calcTop, calcRight, calcBottom）;
    if Checked then
    begin
      canvas.MoveTo（calcLeft, calcTop）;
      canvas.LineTo（calcRight, calcBottom）;
      canvas.MoveTo（calcLeft, calcBottom）;
      canvas.LineTo（calcRight, calcTop）;
    end;
  end;
end;
```

```
procedure Register;
begin
  RegisterComponents（'jxg' ,[TQRCheckBox]）;//将该类注册于 jxg 页上
end;
end.
```

（2）数据明了（data-aware）组件。一个数据明了组件是标准可打印组件和标准数据明了组件两者的混合。然而，照 QuickReport 标准应指向 DataSet 而不是 Datasource。因此，应该为该控件专门设计一个用于编辑 Fieldname 属性的属性编辑器，通过这个属性编辑器来列出 Dataset 的所有字段。

2. 扩展表达式计算器

在表达式计算器中可以扩充自己的函数和计算表达式。事实上，在表达式计算器中的所有函数可以作为函数附件来实现。产生一个附加函数需要完成以下两步：

（1）写一个从 TQREvElementFunction 中继承下来的函数类；

（2）注册该函数。

附加函数有两个基本类型：简单函数和统计函数。这两类函数既可以带参数，也可以不带参数，也可以带数目变化的参数。不同类型函数举例如下：

不带参数的简单类型	Time
带参数的简单类型	Str（<numeric>）
不带参数的统计类型	Count
带参数的统计类型	Sum（<numeric>）

简单函数

实现一个简单函数比较容易，只需从 TQREvElementFunction 生成一个子类并且重载 Calculate 函数即可。Calculate 应该返回函数能返回的任何类型。以时间 Time 函数为例，代码如下：

```
TQREvTimeFunction = class（TQREvElementFunction）
public
  function Calculate : TQREvResult; override;
end;

function TQREvTimeFunction.Calculate : TQREvResult;
begin
  Result.Kind := resString;
  Result.StrResult := TimeToStr（Now）;
end;
```

因为 Time 函数不带有参数，所以我们应该增加一层错误检测程序，代码如下：

```
TQREvTimeFunction = class（TQREvElementFunction）
public
  function Calculate : TQREvResult; override;
end;
function TQREvTimeFunction.Calculate : TQREvResult;
begin
  if ArgList.Count = 0 then
  begin
    Result.Kind := resString;
    Result.StrResult := TimeToStr（Now）;
  end else
    Result.Kind := resError;
  end;
```

其中 ArgList.count 将告诉有多少个参数会传递给函数，并且这些参数值可以通过 Arguments 函数来取得。

第十八章

BDE应用

热点透视

Borland 提供了一套与数据库连接的解决方案，那就是 Borland 数据库引擎。在进行 Delphi 数据库编程时，需要 BDE 的支持，同时 Delphi 也附带了许多控制 BDE 别名的工具。但是，有时为了能够在自己的应用程序中控制这些信息，需要进行一定的编程。本章将围绕着这一主题思想进行阐述，主要内容包括：

- ✧ 在代码中创建 BDE 别名
- ✧ 控制 BDE 别名的信息



在代码中创建 BDE 别名



遇到难题

安装盘中常常会包含 Borland 的数据库引擎（Borland Database Engineer）BDE，有时为了允许用户修改参数，所以将一些参数留给了用户进行配置。如果用户不小心将预设的 BDE 别名删除了，当然希望能通过代码对其进行恢复。也就是说，创建 BDE 别名可以通过自己的程序进行控制。如何才能实现呢？



钥匙在此

调用 TSession.AddAlias（）方法可以为数据库添加别名。该方法定义如下：

procedure AddAlias（const Name, Driver: String; List: TStrings）;

在程序运行时调用该方法可以为 Session 创建一个 BDE 的数据库别名。其中参数 Name 是赋给该别名一个独一无二的名字。Driver 是 SQL Links 数据库驱动使用的名字。List 是创建该别名的参数列表。另外，为了能将创建的数据库别名存储起来，需要调用 SaveConfigFile 方法。

一个 Session 的 ConfigMode 属性决定了创建的别名对于该 Session 来说是本地的，或者对于所有的 BDE 客户端应用程序都是可用的。

TSession.ConfigMode 决定了 BDE 是如何处理用 AddAlias 或 AddStandardAlias 所添加的别名，ConfigMode 也决定了当一个应用程序在调用 GetAliasNames 时，应该为该 Session 包含哪些别名。该属性是一个集合类型，其定义格式为

```
const
cmVirtual = [cfmVirtual];
   cmPersistent = [cfmPersistent];
   cmSession = [cfmSession];
   cmAll = [cfmVirtual, cfmPersistent, cfmSession];
type
   TConfigModes =  （cfmVirtual, cfmPersistent, cfmSession）;
   TConfigMode = set of TConfigModes;
property ConfigMode: TConfigMode;
```

其缺省设置为 cmAll，该值包括了所有可能的集合数值。下表概括了该属性可能的配置

值。

表 18-1　　cmAll 的配置值

模式	含义
cfmVirtual	表示所有 BDE 中的配置文件以及 Session 的本地别名对于该 Session 都可用
CfmPersistent	仅仅 BDE 中的配置文件的别名可用
cfmSession	仅仅由该 Session 创建的别名可用

在一个 Session 中创建的别名不会自动存储到 BDE 的配置文件中，为了将添加的别名存到别名文件中，需要调用 SaveConfigFile 方法。



跟我来

从控件面板 Data Controls 页面中选择 Session 控件且加入到应用程序的窗体中，并给定一个 SessionName，如 Mysession。

清单 18-1 为代码创建数据库别名的例子。

清单 18-1　代码创建数据库别名

```
procedure TForm1.Button1Click（Sender: TObject）;
var
  MyList: TStringList;
begin
  MyList := TStringList.Create;
  try
    with MyList do
    begin
      //数据库别名的定义信息
      Add（'DEFAULT DRIVER=PARADOX'）;
      Add（'ENABLE BCD=FALSE'）;
      Add（'PATH=d:\borland\Borland Shared\Data'）;
    end;
    //添加别名
    Session1.AddAlias（'MyParadox', 'PARADOX', MyList）;
    //将配置信息存储起来
    Session1.SaveConfigFile;
  Finally
    MyList.Free;
  end;
end;
```

再送礼包

另外，如果所创建别名的数据库为 Paradox、dBASE 或者 ASCII 时还可以使用更为简单的方法来创建别名，该方法即为 AddStandardAlias（），其定义格式为：

procedure AddStandardAlias（const Name, Path, DefaultDriver: String）;

其中，参数 Name 与 AddAias（）方法中的 Name 参数一样。参数 Path 为存放表的路径。DefaultDriver 为 Paradox、dBASE 或者 ASCII 中的一种。

例如，以下这个例子可以达到上述同样的效果。.

```
procedure TForm1.BitBtn1Click（Sender: TObject）;
begin
  try
    Session1.AddStandardAlias（'MyParadox',
        'd:\borland\Borland Shared\Data','PARADOX'）;
    Session1.SaveConfigFile;
  finally
     showmessage（'创建错误'）;
  end;
end;
```



该程序运行结束后不会自动将这个别名保存到文件中。该程序运行结束后会将该别名删除。



控制 BDE 别名的信息

遇到难题

使用过 Delphi 数据库的程序员都明白，Borland 提供了一套与数据库连接的解决方案，那就是 Borland 数据库引擎。同时 Delphi 也附带了许多控制 BDE 别名的工具，例如 BDE Administrator 以及 Data Explorer 等数据库工具，这些工具都能够对 BDE 别名进行控制，例如添加别名、删除别名以及编辑参数等等。有时程序员在应用程序中希望能对这些信息进行控制，例如对别名中参数的设置、更改以及读取等等。



钥匙在此

为了能很好的控制 BDE 别名，需要使用 Tsession 类。因为 Session 封装了许多数据库的连接，使用 Session 的属性及其方法可以进行 BDE 别名的操作。每一个与 Session 关联的数据库控件 Tdatabase 都有一个 BDE 别名。数据库别名以及与之关联的 Tsession 的方法有三个主要用途：

（1）确定别名的可视性；

（2）获取别名和数据库驱动程序信息；

（3）产生、修改和删除别名。

以下将详细描述这几方面的功能。

1. 指定别名的可视性

一个 Session 的 ConfigMode 属性决定了 BDE 别名对该 Session 是否可见。这个属性是一个集合类型，描述了 Sessions 中哪一类型是可视的。其使用方法可参见前一节。

2. 使 Session 别名对于其他的 Session 和其他应用程序可见

当一个 Session 产生了一个别名时，BDE 则会在内存中储存该别名的一个拷贝。缺省的情况下这个拷贝对于 Session 来讲都是本地化的。如果创建这些别名的 ConfigMode 属性是 cmAll 或者是 cfmPersistent 的话。同一个程序的其他 Session 仅仅只能够看到该别名。

为了使一个别名对于其他的 Session 或者其他应用程序可用的话，需要使用 Session 的 SaveConfigFile 方法。该方法会将存在内存中的可用别名存储到 BDE 的配置文件中，在那里它们可以被其他的应用程序读取和使用。

3. 确定已知的别名、驱动程序以及参数

Tsession 组件提供了 5 个方法可以使应用程序能获得关于 BDE 别名的信息，包括参数信息和驱动程序的信息。这 5 个方法分别是：

GetAliasNames 列举了该 Session 能访问的所有别名；

GetAliasParams 列举了指定别名的参数；

GetAliasDriverName 返回被别名所使用的驱动程序名的字符串；

GetDriverNames 返回了一列该 Session 可以使用的 BDE 驱动程序名；

GetDriverParams 返回指定驱动程序的驱动参数。

4. 产生、修改和删除别名

一个 Session 能够创建、修改和删除别名。

通过 AddAlias 可以为 SQL 数据库服务器创建一个新的 BDE 别名。使用 AddStandardAlias 可以为 Paradox、dBASE 或者 ASCII 表创建一个 BDE 别名。这两个方法在前一节已经介绍过了。

在创建别名之后，可以通过调用 ModifyAlias 来更改它的参数。ModifyAlias 带有两个参数，一个是要修改的别名的名称，另一个载有要修改的参数以及参数的值。

删除一个以前创建的别名，可以调用 DeleteAlias 方法。该方法带有一个参数，为要删除的别名的名称。DeleteAlias 使一个别名对该 Session 不可用。



如果该别名在此之前已经使用 SaveConfigFile 将它存储在配置文件中，则 DeleteAlias 方法不会将该别名从 BDE 文件中删除。为了在调用 DeleteAlias 之后能从配置文件中删除该别名，需要再一次调用 SaveConfigFile。



跟我来

下面看一个修改 BDE 别名参数的例子。在实现以下代码之前可以先运行清单 18-1 来创建数据库别名“MyParadox”，其数据库为标准桌面数据库，即为 Standard。完整代码参见清单 18-2。

清单 18-2　修改别名参数

```
unit bdeConfigP;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, Buttons, Mask, DBTables;

type
  TForm1 = class（TForm）
    Label1: TLabel;
    Label3: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    Edit_PathName: TEdit;
    Edit_DriverName: TEdit;
    Session1: TSession;
    procedure FormCreate（Sender: TObject）;
    procedure FormShow（Sender: TObject）;
    procedure FormClose（Sender: TObject; var Action: TCloseAction）;
    procedure BitBtn1Click（Sender: TObject）;
    procedure BitBtn2Click（Sender: TObject）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

const
 DataBaseAllias='MyParadox';
var
   SL:TstringList;
{$R *.DFM}

procedure TForm1.FormCreate（Sender: TObject）;
begin
  SL:=TStringList.Create;
end;

procedure TForm1.FormShow（Sender: TObject）;
begin
   //获取指定 BDE 别名的所有参数及其数值
   with Session1 do begin
        if GetAliasDriverName（DataBaseAllias）='STANDARD'then begin
            GetAliasParams（DataBaseAllias,SL）;
            Edit_PathName.Text:=SL.Values['PATH'];
            Edit_DriverName.Text:=SL.Values['DEFAULT DRIVER'];
        end;
   end;
end;

procedure TForm1.FormClose（Sender: TObject; var Action: TCloseAction）;
begin
 Sl.Free;
end;

procedure TForm1.BitBtn1Click（Sender: TObject）;
begin
   //将参数的修改值存入到配置文件中
   if Session1.GetAliasDriverName（DataBaseAllias）='STANDARD'then begin
       SL.Values['PATH']:=Edit_PathName.Text;
       SL.Values['DEFAULT DRIVER']:=Edit_DriverName.Text;
```

```
        Session1.ModifyAlias（DataBaseAllias,SL）;
        Session1.SaveConfigFile;
        showmessage（'修改成功！'）;
        close;
    end;
end;

procedure TForm1.BitBtn2Click（Sender: TObject）;
begin
  Close;
end;

end.
```

该程序代码只读取和更改类型为 standard 的标准本地数据库的别名。在窗体 show 时如果指定的‘MyParadox’别名为本地库时，则将参数及其数值读取出来，并将相应的主要参数值进行显示。按确定按钮后则将需要修改的参数值进行修改并存储起来。运行该代码，其界面如图 18-1 所示。



图 18-1　修改 BDE 别名的参数

第十九章

数据库

热点透视

在编写数据库应用程序时，有很多内容需要讲述。由于篇幅的限制，本章主要解答程序员经常碰到的一些问题。例如如何自动登录数据库，讲述如何通过程序中内置用户的登录信息进行数据库的连接；如何动态创建数据表格，讲述如何由代码动态控制数据库表；如何将文件存入到数据库中，讲述如何将图形文件和声音文件存入到数据库中，以及如何利用显示这些数据；如何实现计算字段，讲述如何使用字段编辑器来添加计算字段。本章内容主要包括：

- 自动登录数据库服务器
- 动态创建数据库表格
- 将文件存入数据库中
- 实现计算字段



自动登录数据库服务器



遇到难题

在进行数据库服务器登录时，特别是 SQL 数据库时（例如常用的有 Oracle、Sysbase、SQL Server 等数据库服务器）需要进行安全性检查，如需要检查用户名和用户密码。因此，如果 Delphi 数据库应用程序要与这些数据库进行连接时同样需要告诉服务器用户名和用户密码。缺省的情况下，在进行数据库连接时，会执行语句 Table1.Open 以弹出以下对话框，如图 19-1 所示，只有输入合法的用户名和用户密码时才可以实现连接。

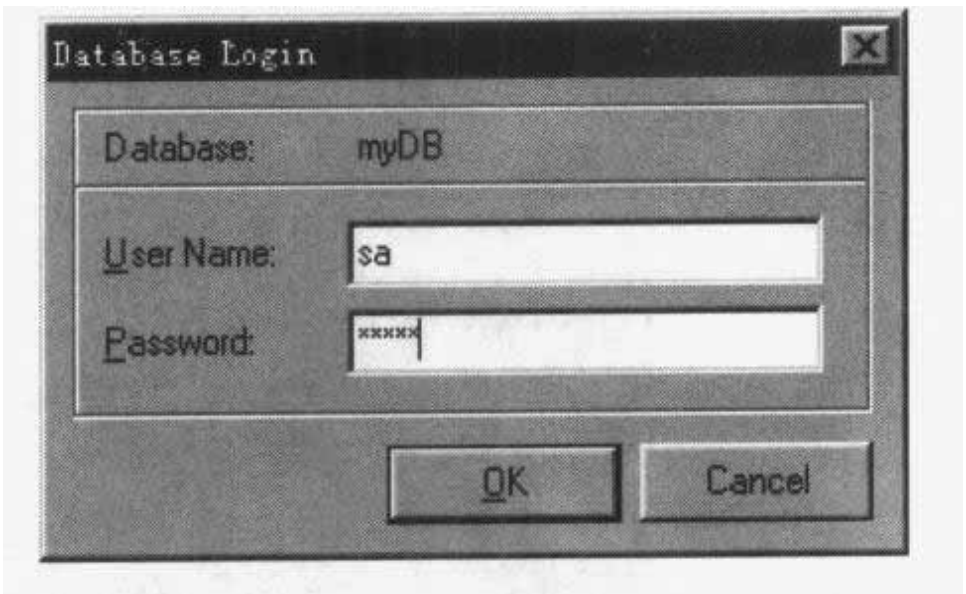


图 19-1　缺省登录对话框

但是，应用程序中登录的用户，用的往往是自己的一套用户管理机制，并不需要使用 SQL 数据库本身的用户，此时，应用程序需要将这一登录过程自动进行，而并不需用户重新录入 SQL 数据库的用户账号。如何将 SQL 数据库的登录账号内置呢？



钥匙在此

Delphi 中包含有 Database 元件和对象，Database 对象具有属性:Params 和 LoginPrompt，充分使用这两个属性可以自动 Login 数据库。下面的例程用于自动 Login。

为了实现自动登录，首先应当将内置的用户登录信息赋值给该对象相应的参数。例如，对于 SQL Server 来说，需要设置以下两个数据库的参数。

```
Database1.Params.Values['USER NAME']:='sa';
Database1.Params.Values['PASSWORD']:='admin';
```

前者为用户名，后者为相应的用户密码。

为了不对用户进行提示，还需要设置数据库对象的 LoginPrompt 属性，该属性指定了当与数据库服务器建立连接时是否显示一个标准的对话框，如图 19-1 所示。为了不出现该对话框，应当将 LoginPrompt 属性设置为 False。



跟我来

下面看一个自动登录数据库服务器的例子。

新建一程序（Project），然后在 Form 上放置 Table、Database、DataSource 以及 DBGrid 各一个，并通过 Object Inspector 将 DBGrid 的 DataSource 指向 DataSource，DataSource 的 DataSet 指向 Table。接着双击 Form，便可进行 Form1 的 Create 事件程序代码编辑，输入以下代码：

```
Database1.Params.Values['USERNAME']:='sa';
Database1.Params.Values['PASSWORD']:='admin';
Database1.LoginPrompt:=False;
Database1.DatabaseName:='MYDB';
Database1.AliasName:='MyDatabase';
Table1.DatabaseName:='MYDB';
Table1.TableName:='EMPLOYEE';
Table1.Open;
```

运行该程序便可以自动登录数据库服务器，并将数据表中的数据显示在 DBGrid 中。对于不需要用户名和密码的数据库驱动，可以设置 USERNAME 和 PASSWORD 为空。



动态创建数据库表格



遇到难题

在目前很多通用性程度较高的数据库软件中往往提供了动态创建数据表格的功能。例如，若应用程序需要制作商业性安装盘的话，则基于数据库的应用程序需要支持动态创建数据库表格的功能。当然，很多数据库服务器本身带有支持创建数据对象的脚本，即当在该数据库中创建了很多数据表和存储过程时，可以将这些表格和存储过程的创建过程存成一个脚本文件，然后在新环境安装时，再创建这些表格和存储过程时就不需手工逐个创建了，而只需通过运行存储过的脚本文件即可实现数据库环境的建立。这种方式具体可以参考相关数据

库的书籍。

但是，有时要求应用程序能根据不同的要求创建不同的表格和存储过程，这就需要应用程序具有支持动态创建数据表格和存储过程的功能。



钥匙在此

为了动态创建数据库表格，可以使用 Ttable 控件来实现。通过该控件实现动态创建表需要经过定义字段、定义索引和创建表格这三个步骤。

1. 定义字段

定义数据表的字段需要使用 Ttable 控件的 FieldDefs 属性，该属性本身又是一个 TFieldDefs 对象。TfieldDefs 对象提供了 AddFieldDef（）和 Add（）方法用于在定义表时添加字段。其中为了向前兼容 Add（）方法还在使用，一般可以使用 AddFieldDef（）。现在分别说明如下。

Add（）方法

定义格式：procedure Add（const Name: String; DataType: TFieldType; Size: Integer = 0; Required: Boolean = False）；

描述：创建一个新字段对象，并将它添加到 TfieldDefs 对象的 Items 属性中。其中参数 Name 表示字段名；DataType 为字段的类型，例如 ftstring 表示字符串类型；Size 为字段的大小，对于非字符类型字段的大小指定为 0；Required 确定该字段是否必须包含数值，如果为 True，则表示该字段必须带有值。

例子：Add（'用户代码'， ftString， 20， true）;表示为该表添加了“用户代码”字段，类型为字符串，大小为 20，必须包含值。

AddFieldDef（）方法

定义格式：function AddFieldDef: TFieldDef;

描述：其功能与 Add（）方法类似。该函数将返回一个新的 TfieldDef 对象。在调用该函数之后，应该设置新添加字段对象的属性，包括 name、typem、size 以及 Required 属性。

例子：

```
with AddFieldDef do begin
  Name := '用户代号';
  DataType := ftstring;
  Size:=20;
  Required := True;
end;
```

其作用与上述的例子一样。

2. 定义索引

为了提高数据库的性能，需要定义数据表的索引。可以使用 Ttable 控件的 IndexDefs 属性来定义数据表的索引，该属性本身也是一个对象，即 TindexDefs 对象。TIndexDefs 对象也提供了类似 TfieldDefs 对象的两个方法，即 AddIndexDef（）和 Add（）方法。这两个方法用于定义表的索引。其中，一般可以使用 AddindexDef（）方法来定义索引信息。现在分

别说明如下。

Add（）方法

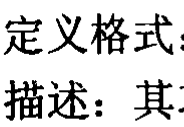
定义格式：procedure Add（const Name， Fields: String; Options: TIndexOptions）；

描述：创建一个新索引对象，并将之添加到 TindexDefs 对象的 Items 属性中。

参数：其中参数 Name 为索引名；Fields 为该索引所包含的一列字段，各字段间使用分号隔开，例如“code;name”；Options 描述了索引的特征。

例子：IndexDefs.Add（'idx_user'， '用户代码'， [ixPrimary， ixUnique]）；

表示为该表的用户代码添加了“idx_user”索引。



Options 为 TIndexOptions 类型，说明如下：

```
type
    TIndexOption = （ixPrimary， ixUnique， ixDescending， ixCaseInsensitive，
ixExpression， ixNonMaintained）；
    TIndexOptions = set of TIndexOption;
```

这是一个集合类型，可以取空，也可以取表 19-1 中值的组合。

表 19-1　　TindexOptions 值描述

值	描述
IxPrimary	数据表的主索引
IxUnique	表示索引的每个值都是独一无二的
ixCaseInsensitive	索引排序对大小写敏感

注意，如果产生了一个表类型不支持的索引时将会出现异常。

AddindexDef（）方法

定义格式：function AddIndexDef: TIndexDef;

描述：其功能与 Add（）方法类似。该函数将返回一个新的 TindexDef 对象。在调用该函数之后，应该设置新添加字段对象的属性，包括 name、fields、options。

例子：

```
with AddindexDef do begin
  Name := 'idx_user';
  Fields := '用户代号';
  Options:= [ixPrimary， ixUnique];
end;
```

其作用与上述的例子一样。



如果需要为已存在的表添加索引，需要使用 AddIndex（）方法。

3. 创建表格

当定义完字段以及索引后可以调用 Ttable.CreateTable 方法来创建表格。如果所创建的表已经存在，该方法会覆盖已经存在的表结构以及数据。为了避免这种情况的发生，在调用

该方法之前应该检查 Exists 属性。该属性是只读的，用于确定当前指定的表是否存在。



跟我来

清单 19-2 代码将完整地说明动态创建表的整个过程。

清单 19-2　动态创建表

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, DBTables, StdCtrls;

type
  TfrmMain = class（TForm）
    Button1: TButton;
    tmptable: TTable;
    procedure Button1Click（Sender: TObject）;
  private
    { Private declarations }
    procedure CreateATable（STName : String）;
  public
    { Public declarations }
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.DFM}
const
    DB_Name='myData';

procedure Tfrmmain.CreateATable（STName : String）;
begin
      with TmpTable do
```

```
begin
    Active := False;
  //设置表名
    DataBaseName := DB_NAME;
    TableName := STName;
    TableType := ttDefault;
     //如果要创建的表已经存在，则不进行创建
    if exists then begin
        showmessage（'表已经存在!'）;
        exit;
    end;
    //定义字段
with FieldDefs do
begin
      Clear;
      Add（'用户代码'， ftString， 20， true）;//字符串类型
      Add（'用户名'， ftString， 8， False）;
      Add（'生日'， ftdate， 0， False）;//日期类型
      Add（'工资'，ftCurrency，0，False）;//货币类型
end;
//定义索引
IndexDefs.Clear;
IndexDefs.Add（'idx_'+STName， '用户代码'， [ixPrimary， ixUnique]）;
//创建表格
try
   CreateTable;
   except on EDBEngineError do begin
            showmessage（tablename+'表创建错！'）;
            exit;
        end;
    end;
end;

procedure TfrmMain.Button1Click（Sender: TObject）;
begin
    CreateATable（'user'）;
end;

end.
```

该代码中为了创建表而定义了 CreateATable（）过程。在创建过程中，首先给定要创建的表名，然后检测该表是否已经存在。如果已经存在，则不予创建。当可以创建表时，先定义该表的字段，再定义索引，当这些基本信息定义完后，调用 Table 的 CreateTable（）方法来创建数据表。



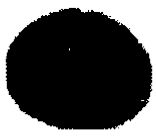
再送礼包

当然，还可以通过动态的 SQL 语句来动态创建数据表。例如：

```
with query1 do begin
    close;
    SQL.clear;
    SQL.ADD（‘CREATE TABLE user ）
    SQL.ADD（‘用户代号 char（5）’,
    SQL.ADD（‘用户名称 char（10） NULL）’）
    TRY
        EXECSQL;//执行 SQL 语句创建表格
    EXCEPT
        Showmessage（‘创建表失败！’）;
    END
end;
```



将文件存入数据库中



遇到难题

有时用户希望自己的数据能图、文、声并茂，也就是使数据库能支持多媒体功能。例如，如果编写的应用程序是用来管理动物资料的，用户则希望将动物的种类、名称、图片以及动物的叫声都能存入到数据库中。于是在调用某一动物资料的时候，能看到动物的图片，也能听到动物的声音。也许，您认为用户的要求不合理，觉得数据库怎么能支持多媒体功能呢？其实，数据库并不支持多媒体功能，多媒体功能应该属于应用程序。而对于用户来讲，数据库加上程序就可以实现多媒体功能了。通过数据库存储信息和程序向用户展示这些信息。这样的话，就需要应用程序能支持将多媒体文件存入和导出的功能。



钥匙在此

这些多媒体文件一般都比较大，因此，需要有一个能存放大文件的字段类型来支持。这就是 BLOB 字段。

BLOB（Binary Large OBject）字段用来保存不定量的数据。例如，一个记录中的 BLOB 字段可能包含了 3 个字节的数据，而另一个记录则可能包含 3K 的字节数据。BLOB 字段最适合于保存大容量的文本、图像以及诸如 OLE 对象那样的数据流。

Delphi 中有一个 Tfield 的派生对象——TblobField，它封装了 BLOB 字段。TblobField 有一个类型为 TblobType 的属性 Blobtype，该属性表示保存在 Blob 字段中的数据类型。在 DB 单元中，该类型是这样声明的：

```
type TBlobType = ftBlob..ftOraClob;
```

这是一个枚举类型，所有的字段类型和与之对应的数据类型如表 19-2 所列。

表 19-2　　TblobField 字段类型和相应的数据类型

字段类型	数据类型描述
ftBlob	BLOB 字段
ftMemo	文本 Memo 字段
ftGraphic	位图字段
ftFmtMemo	格式文本 Memo 字段
ftParadoxOle	Paradox OLE 字段
ftDBaseOle	dBase OLE 字段
ftTypedBinary	类型二进制字段
ftOraBlob	在 Oracle8 表中 BLOB 字段
ftOraClob	在 Oracle8 表中 CLOB 字段

通过 TblobField 读写数据时，大部分工作是把文件加载到 Blob 中或者把该字段中的内容保存到本地文件中。

另外，也可以使用 TblobStream 对象对 Blob 字段进行操作。TblobStream 是一个流对象，它提供了应用程序从 Blob 字段中读写数据的服务，即提供了操作 Blob 字段的流机制。



跟我来

下面这个程序改造了 Delphi 中的一个演示程序（原演示程序在\Borland\Delphi5\Demos\Db\Fishfact 下），使该程序除了能显示鱼的图片外，还能够播放相应的声音。

该程序的数据库结构如下所示：

字段	字段类型	字段大小	描述
Name	Char	20	存放名字
Image	Blob		存放图形文件
Wave	Blob		存放声音文件

创建一个主窗体，它所包含的元件如图 19-2 所示。使用 Fields Editor 可以添加 Blob 字段类型组件。如下例中的 tblSoundsImage、tblSoundsWave 都为 Blob 字段组件。

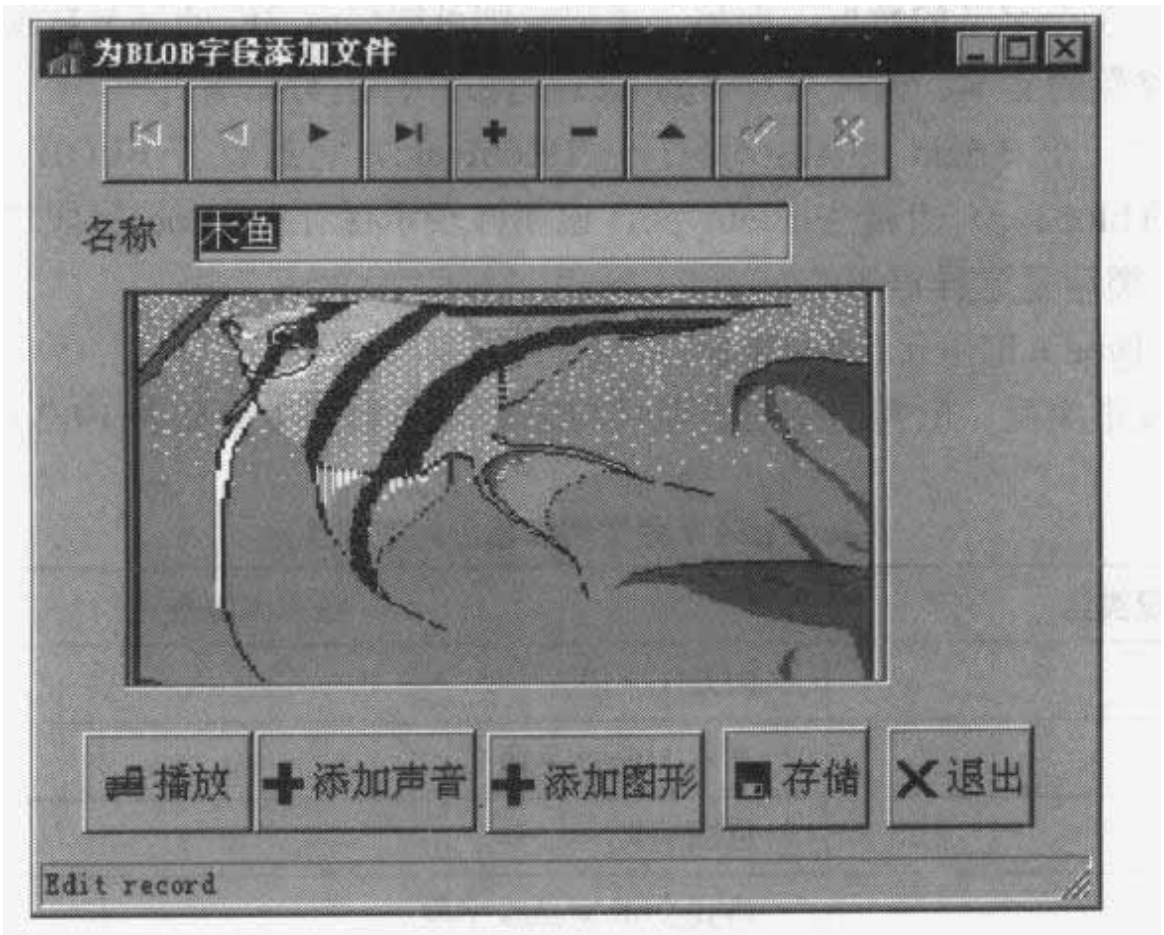


图 19-2　为 Blob 字段添加文件

添加按钮用于向数据库表中存放图形和声音文件。对应于该按钮的单击事件是这样处理的：

```
//添加声音文件
procedure TMainForm.sbAddClick（Sender: TObject）;
begin
  if OpenDialog.Execute then
  begin
    //当 Table 处于添加或编辑状态时，则向数据库表中添加文件
    if （tblSounds.state=dsInsert） or （tblSounds.state=dsEdit） then
    tblSoundsWave.LoadFromFile（OpenDialog.FileName）;
  end;
end;

//添加图形文件
procedure TMainForm.SpeedButton_ImageClick（Sender: TObject）;
begin
```

```
    if OpenDialog.Execute then
    begin
      //当 Table 处于添加或编辑状态时，则向数据库表添加文件
      if （tblSounds.state=dsInsert） or （tblSounds.state=dsEdit） then
      tblSoundsImage.LoadFromFile（OpenDialog.FileName）;
    end;
  end;
```

上述代码用于存放声音文件和图形文件。首先执行的是 OpenDialog。如果这个操作成功，并且当前的 tblSounds 处于添加或者编辑状态的话，则使用 TblobField 的 LoadFromFile（）方法将文件存入到相应的 Blob 字段中。

类似的，存储按钮用于把当前 Wave 字段中的声音数据保存到一个外部文件中。对应于该按钮的代码为：

```
procedure TMainForm.sbSaveClick（Sender: TObject）;
begin
  with SaveDialog do
  begin
    //如果当前 Blob 字段的数据不为空时则将数据存成外部文件
    if Execute and （not tblsoundsWave.isnull） then
      tblSoundsWave.SaveToFile（'c:\temp\a.wav'）; // 将 wave 文件存成外部文件
  end;
end;
```

播放按钮的作用是负责从 Blob 字段中读出 Wave 数据，并传递给 API 函数 PlaySound()，这样就能播放声音了。对应于该按钮的事件句柄代码如下：

```
procedure TMainForm.sbPlayClick（Sender: TObject）;
var
  B: TBlobStream;
  M: TMemoryStream;
begin
B := TBlobStream.Create（tblSoundsWave， bmRead）; // 创建 Blob 流
Screen.Cursor := crHourGlass;                    // 更换光标成 hourglass
try
  M := TMemoryStream.Create;                     // 创建 memory 流
  try
    M.CopyFrom（B， B.Size）;                    // 从 Blob 流复制到 memory 流中
    // 播放声音，如果有错则发生异常
    Win32Check（PlaySound（M.Memory， 0， SND_SYNC or SND_MEMORY））;
  finally
    M.Free;
  end;
finally
```

```
        Screen.Cursor := crDefault;
        B.Free;
      end;
    end;
```

在这个事件句柄中，首先是创建一个 Tblobstream 实例。传递给 TBlobstream.create（）的第一个参数是 Blob 字段对象，第二个参数则指明是否要打开流，并以何种方式打开。一般情况下，对 Blob 的访问是只读的，这时要用 bmRead 作为参数。如果既要可读又要可写，则要用 bmReadWrite 参数。

接着，创建一个 TmemoryStream 的实例 M，这时光标的形状会发生变化，表示此操作需要数秒才能完成，然后把流 B 复制给流 M。

用来播放 Wave 声音的函数为 Playsound（），它需要传递一个文件名或一个内存指针作为第一个参数，但 Tblobstream 不支持以指针方式访问流数据，而 TmemoryStream 却可以通过它的 Memory 特性来做到这一点。这样，就可以通过 PlaySound（）来播放 M.Memory 指定的数据。调用了 PlaySound（）后，需要释放流并恢复光标的形状。该程序的主要源代码见清单 19-3。

清单 19-3　Blob 字段的赋值与读取

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, DBCtrls, DB, DBTables, StdCtrls, Mask, Buttons, ComCtrls;

type
  TMainForm = class（TForm）
    tblSounds: TTable;
    dsSounds: TDataSource;
    edFileName: TDBEdit;
    Label2: TLabel;
    OpenDialog: TOpenDialog;
    SaveDialog: TSaveDialog;
    stbStatus: TStatusBar;
    DBImage1: TDBImage;
    tblSoundsName: TStringField;
    tblSoundsImage: TBlobField;
    tblSoundsWave: TBlobField;
    sbExit: TSpeedButton;
    sbSave: TSpeedButton;
```

```
    SpeedButton_Image: TSpeedButton;
    sbAdd: TSpeedButton;
    sbPlay: TSpeedButton;
    dbnNavigator: TDBNavigator;
    procedure sbPlayClick（Sender: TObject）;
    procedure sbAddClick（Sender: TObject）;
    procedure sbSaveClick（Sender: TObject）;
    procedure sbExitClick（Sender: TObject）;
    procedure FormCreate（Sender: TObject）;
    procedure SpeedButton_ImageClick（Sender: TObject）;
  private
    procedure OnAppHint（Sender: TObject）;
    end;

  var
    MainForm: TMainForm;

  implementation

  {$R *.DFM}

  uses MMSystem;

  procedure TMainForm.sbPlayClick（Sender: TObject）;
  var
    B: TBlobStream;
    M: TMemoryStream;
  begin
  B := TBlobStream.Create（tblSoundsWave，  bmRead）; // 创建 Blob 流
  Screen.Cursor := crHourGlass;                    // 更换光标成 hourglass
  try
    M := TMemoryStream.Create;                     // 创建 memory 流
    try
      M.CopyFrom（B，  B.Size）;                   // 从 Blob 流复制到 memory 流中
      // 播放声音，如果有错则发生异常
      Win32Check（PlaySound（M.Memory，  0，  SND_SYNC or SND_MEMORY））;
    finally
      M.Free;
    end;
  finally
```

```
      Screen.Cursor := crDefault;
      B.Free;
    end;
end;

//添加声音文件
procedure TMainForm.sbAddClick（Sender: TObject）;
begin
  if OpenDialog.Execute then
  begin
    //当 Table 处于添加或编辑状态时，则向数据库表中添加文件
    if （tblSounds.state=dsInsert） or （tblSounds.state=dsEdit） then
    tblSoundsWave.LoadFromFile（OpenDialog.FileName）;
  end;
end;

//添加图形文件
procedure TMainForm.SpeedButton_ImageClick（Sender: TObject）;
begin
  if OpenDialog.Execute then
  begin
    //当 Table 处于添加或编辑状态时，则向数据库表中添加文件
    if （tblSounds.state=dsInsert） or （tblSounds.state=dsEdit） then
    tblSoundsImage.LoadFromFile（OpenDialog.FileName）;
  end;
end;

procedure TMainForm.sbSaveClick（Sender: TObject）;
begin
  with SaveDialog do
  begin
    //如果当前 Blob 字段的数据不为空时则将数据存成外部文件
    if Execute and （not tblsoundsWave.isnull） then
      tblSoundsWave.SaveToFile（'c:\temp\a.wav'）; // 将 wave 文件存成外部文件
  end;
end;

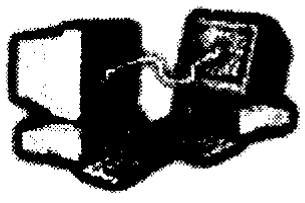
procedure TMainForm.sbExitClick（Sender: TObject）;
begin
  Close;
```

```
end;

procedure TMainForm.FormCreate（Sender: TObject）;
begin
  Application.OnHint := OnAppHint;
end;

procedure TMainForm.OnAppHint（Sender: TObject）;
begin
  stbStatus.SimpleText := Application.Hint;
end;

end.
```



实现计算字段



遇到难题

计算字段是字段编辑器 Fields Editor 中最有价值的成果之一。可以用这些字段来实现各种不同的目的。有两点需要说明：

如果需要使用数据集的两个或多个字段来进行计算，并在另一个字段中显示结果，此时可以使用计算字段。

如果在查询数据库的同时，想完成计算或显示涉及至少一个附加数据集的数据，可使用字段编辑器和计算字段在第一个数据集的新字段中显示结果。但一般来说，虽然计算字段可以实现多种功能，但应该在计算字段内计算，而在查找字段内查阅。

如何实现计算字段呢？



钥匙在此

要实现计算字段，需要经过两步：先使用字段编辑器添加计算字段，然后，在数据集的 OnCalcFields 事件中为可计算字段进行计算赋值。下面进行详细阐述。

1. 使用字段编辑器 Fields Editor 添加计算字段

利用字段编辑器，可以让您在使用数据集中的字段时有更强大的控制功能和灵活性。在字段编辑器中，可以查看一个特定的数据集中所包含的字段。要打开字段编辑器，只要用鼠标双击 Ttable、Tquery 或 TstoredProc 元件，或者选择这些元件，然后在弹出的菜单中选择“Fields Editor”命令即可。在字段编辑器中，可以选择要操作的字段对象，并可以新建计算字段和查找字段。上述操作均可以通过单击鼠标右键调出本地菜单后进行。字段编辑器窗口以及它的本地菜单如图 19-3 所示。

2. 为计算字段进行计算赋值

计算字段的值是通过其他字段或者附加的信息计算而得到的。那么，什么时候开始进行计算呢？Delphi 提供了一个事件句柄用于为计算字段赋值提供编程接口。这一接口即为 TDataSet.OnCalcFields，其定义格式如下：

type TDataSetNotifyEvent = procedure（DataSet: TDataSet） of object;

当 Tdataset.AutoCalcFields 设为 True 时，在下述四种情况下将激发该事件：

（1）该数据集打开时；

（2）数据集处于编辑状态时；

（3）输入焦点从一个数据明了（data-aware）控件移动到另一个数据明了控件时；

（4）从数据库中获得记录时。

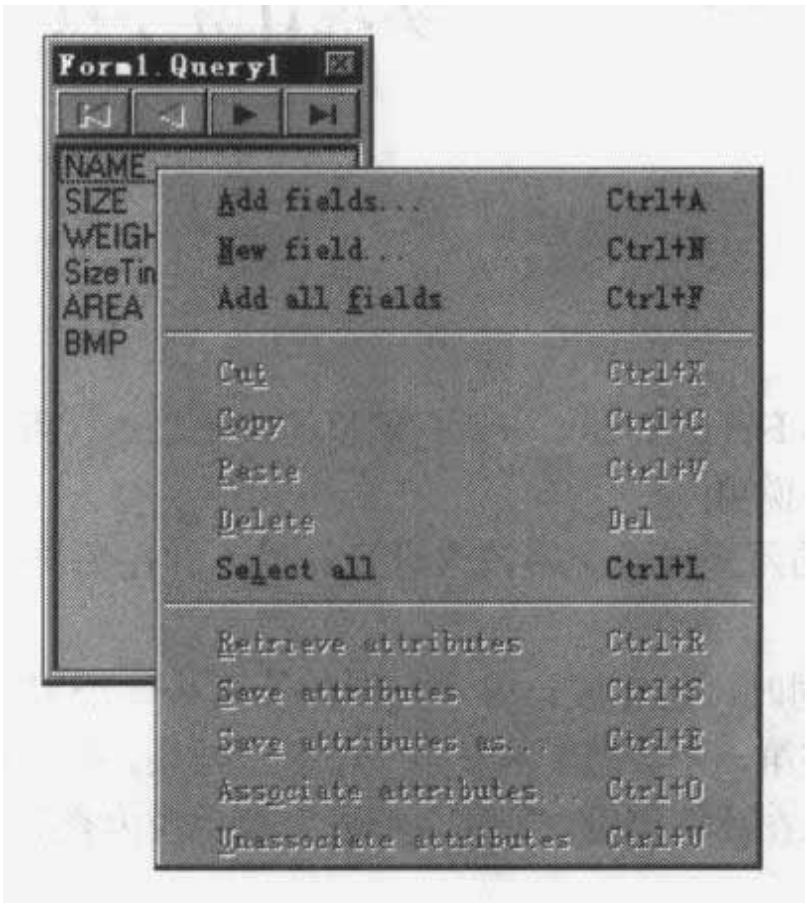


图 19-3　字段编辑器与本地菜单



跟我来

下面举一个实际的例子来说明如何设置和使用计算字段。该例子根据学生所获得成绩自动进行判断成绩的等级。该程序所使用的数据表结构如下表：

表名：**Grade.db**

字段	字段类型	字段大小	描述
学生姓名	Char	8	存放名字
成绩	Integer		学生所获得的成绩

第一步：创建应用程序，然后为主窗体上增加两个 Ttable 控件、TDBGrid 以及 TDBNavigator 控件。

第二步：设置相关的数据库属性。

```
Table1.databasename:= 'mydata';//为表所存放的数据库别名
Table1.tablename:= 'grade.db';
Datasoure1.dataset:=table1;
DBGrid1.datasource:=datasoure1;
```

第三步：为 Table1 添加计算字段。先选择 table1，然后点击鼠标右键来选择菜单“Fields Editor...”，并在该对话框上选择右键菜单“add all fields”，最后向字段编辑器 Fields Editor 中加入所有字段。结果如图 19-4 所示。

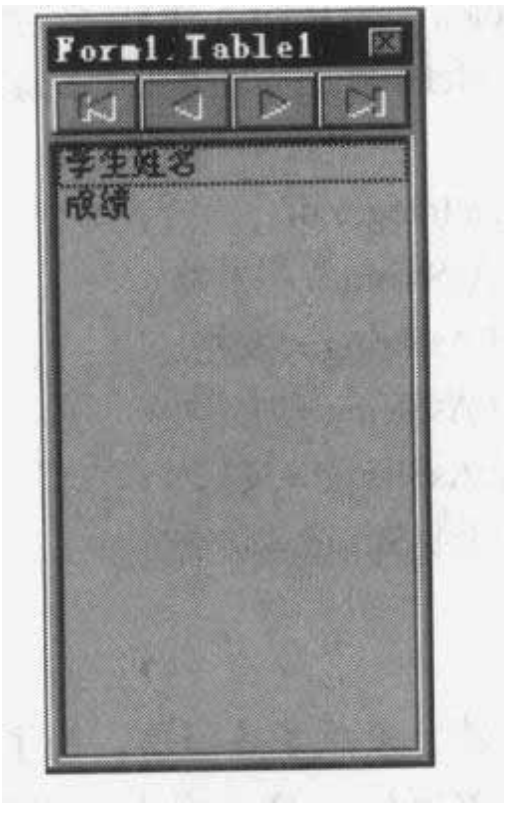


图 19-4　Fields Editor 对话框

在该对话框中点击鼠标右键，然后选择“new field”菜单，此时将弹出新字段定义对话框，如图 19-5 所示。

接着是对新定义的字段进行设置的过程：

（1）在“Name”编辑框中敲入计算字段的名称。注意不要敲入已经存在的字段名。本例为“成绩等级”；

（2）为组件给定一个名字。本例为 Table1FieldLevel；

（3）从“Type”组合框中选择字段的数据类型。本例为 string；

（4）在“size”编辑框中输入字段的大小。大小只在数据类型为 TstringField、TBytesField 和 TvarBytesField 时才有效。本例设置为 8；

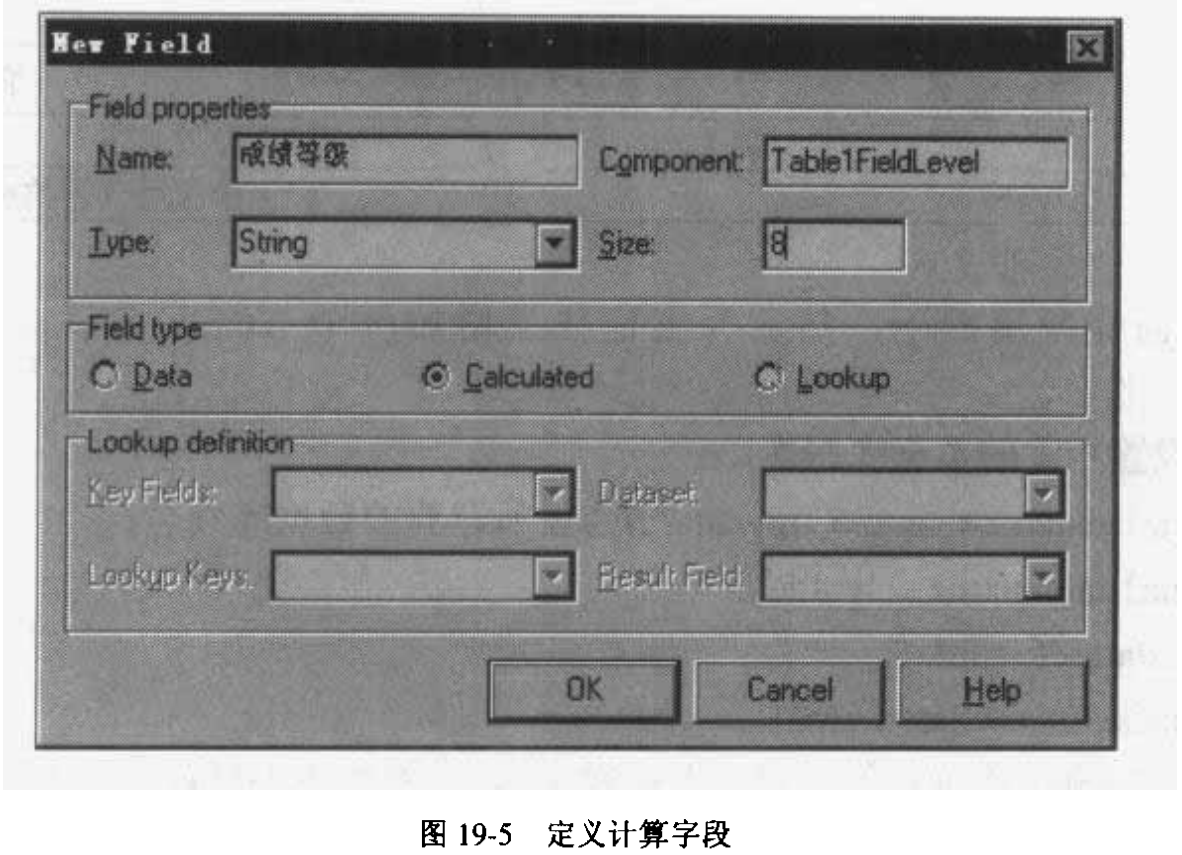


图 19-5　定义计算字段

（5）在“Field Type”选择框中选择“Caculated”类型。

第四步：在 Table. OnCalcFields 事件中写入计算字段的赋值计算代码。本例代码如下：

```
procedure TFrmMain.Table1CalcFields（DataSet: TDataSet）;
begin
      case Table1FieldGrade.AsInteger of
      0..59:Table1FieldLevel.AsString:='不及格';
      60..69:Table1FieldLevel.AsString:='及格';
      70..79:Table1FieldLevel.AsString:='中等';
      80..89:Table1FieldLevel.AsString:='良好';
      90..100:Table1FieldLevel.AsString:='优秀';
      end;
end;
```

该代码根据成绩相应地对计算字段赋予等级值。为了增强显示效果，则根据成绩相应地改变记录的显示颜色。进行 DBGrid1.OnDrawColumnCell（）编程便可以实现改变记录缺省的颜色值，其编程代码如下：

```
procedure TFrmMain.DBGrid1DrawColumnCell（Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState）;
begin
    with TCustomDBGridCracker（sender） do begin
     if table1.FieldByName（'成绩'）.asinteger<60 then
        Canvas.Font.Color := clred
     else
        Canvas.Font.Color := clBlack;
     //根据上面设置的颜色值绘制单元格
     DefaultDrawColumnCell（Rect， DataCol， Column， State）;
```

```
    end;
end;
```

说 明 至于如何改变 DBGrid 元件的缺省颜色，请参见 DBGrid 相应的章节。

第五步：激活 Table1 组件，并运行程序。运行结果如图 19-6 所示。图中在输入了学生的成绩后，会自动根据录入的成绩进行计算，从而判断学生的成绩等级。当获得的成绩不及格时会以红色字体显示这条记录。完整代码请见清单 19-4。

清单 19-4　实现计算字段

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  Db, ExtCtrls, DBCtrls, Grids, DBGrids, DBTables;

type
  TCustomDBGridCracker = class（TCustomDBGrid）;
  TFrmMain = class（TForm）
    Table1: TTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    DBNavigator1: TDBNavigator;
    Table1FieldName: TStringField;
    Table1FieldGrade: TSmallintField;
    Table1FieldLevel: TStringField;
    procedure Table1CalcFields（DataSet: TDataSet）;
    procedure DBGrid1DrawColumnCell（Sender: TObject; const Rect: TRect;
      DataCol: Integer; Column: TColumn; State: TGridDrawState）;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  FrmMain: TFrmMain;

implementation
```

```
{$R *.DFM}

procedure TFrmMain.Table1CalcFields（DataSet: TDataSet）;
begin
        case Table1FieldGrade.AsInteger of
        0..59:Table1FieldLevel.AsString:='不及格';
        60..69:Table1FieldLevel.AsString:='及格';
        70..79:Table1FieldLevel.AsString:='中等';
        80..89:Table1FieldLevel.AsString:='良好';
        90..100:Table1FieldLevel.AsString:='优秀';
        end;
end;

procedure TFrmMain.DBGrid1DrawColumnCell（Sender: TObject;
   const Rect: TRect; DataCol: Integer; Column: TColumn;
   State: TGridDrawState）;
begin
     with TCustomDBGridCracker（sender） do begin
      if table1.FieldByName（'成绩'）.asinteger<60 then
         Canvas.Font.Color := clred
      else
         Canvas.Font.Color := clBlack;
      //根据上面设置的颜色值绘制单元格
      DefaultDrawColumnCell（Rect， DataCol， Column， State）;
     end;
end;
end.
```

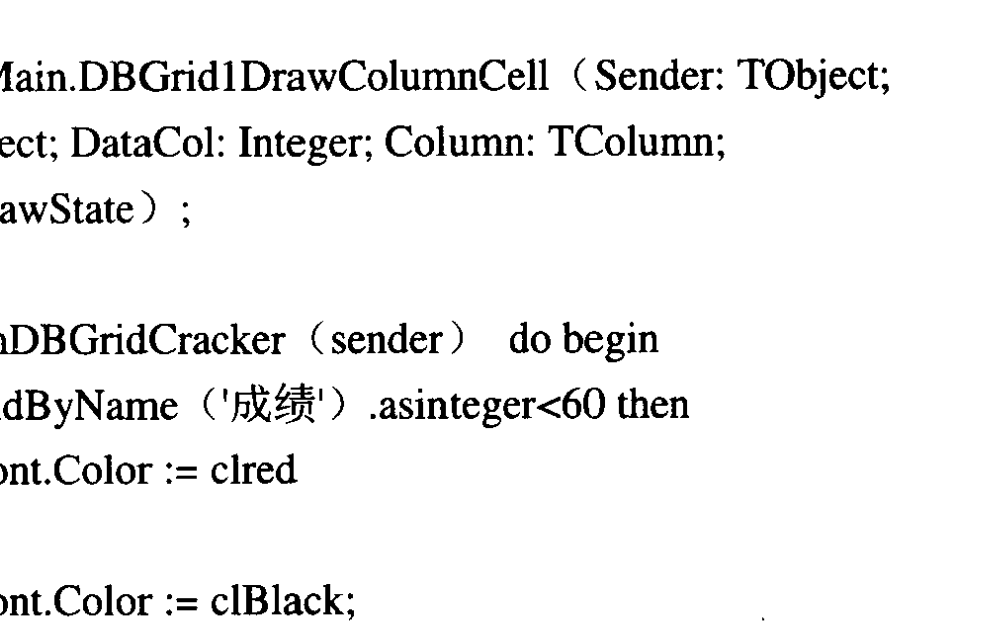


图 19-6　计算字段

第二十章

MIDAS 技术

热点透视

COM、OLE 和 Internet/Intrant 的发展，以及它们之间的联系，刺激了 Windows 的开发市场。其中，最受关注的是多层次和分布式瘦客户数据库应用程序。为了开发者自己，或者就是为了它们的用户，众多的开发者都在寻找着各种方式进行开发。为此，Delphi 4 和 Delphi 5 提供了较为完美的 MIDAS（Multi-tier Distributed Application Services Suite）技术来解决这些问题，而且它将该技术封装成构件，同时将已经熟悉的数据库访问构件集合在一起，从而提供了轻松实现瘦客户和多层应用程序的方法。本章内容有：

- 认识 MIDAS 技术的特点
- 认识 MIDAS 技术
- 创建应用服务器
- 创建客户端应用
- 定制应用服务器
- 创建 ActiveX 控件来分发客户应用程序



认识 MIDAS 技术的特点



遇到难题

COM、OLE 和 Internet/Intrant 的发展，以及它们之间的联系，刺激了 Windows 的开发市场。其中，最受关注的是多层次和分布式瘦客户数据库应用程序。为了开发者自己，或者就是为了它们的用户，众多的开发者都在寻找着各种方式进行开发。为此，Delphi4 和 Delphi5 提供了较为完美的 MIDAS（Multi-tier Distributed Application Services Suite）技术来解决这些问题，而且它将该技术封装成构件，同时将已经熟悉的数据库访问构件集合在一起，从而提供了轻松实现瘦客户和多层应用程序的方法。



钥匙在此

1. 一个简单的三层模式

一个多层客户/服务器应用程序被分成了若干个独立机器。多层应用程序通过局域网甚至 Internet 共享数据并互相通信。这种方式具有较多的特点和优点，如中心服务和瘦客户应用等。

多层应用程序最简单的形式为“三层模式”（three-tiered model），它由三个部分组成：

- 客户应用程序（client application）存在于用户计算机上，实现用户接口；
- 应用程序服务器（Application server） 存在于各客户端可访问的计算机上，它提供中心服务和数据服务；
- 远程数据库服务器（Remote database server） 提供关系数据库管理系统（RDBMS）。

在这种三层结构模式中，应用程序服务器管理客户和远程数据库服务器之间的数据流，所以有时将其称之为“数据经纪人”（data broker）。对于 Delphi 开发人员来说，在应用程序的创建时，仅仅需要开发前两层（即客户应用程序和应用程序服务器）第三层一般是选择第三方的产品，如 Oracle、Sysbase、MS SQL Server 以及 DB2 等等，当然，Inprise 本身的 InterBase 也可以提供较好的数据库功能。

Delphi 对多层应用的支持是利用多层分布式应用服务组（MIDAS）技术来实现的。本章主要讨论如何使用 MIDAS 来创建多层数据库应用。一旦掌握如何创建和管理三层应用的话，你就能够根据你所需要的应用来创建和增加其他服务层。

2. 多层应用模式的优点

多层应用模式将数据库应用分成若干个逻辑模块。客户应用程序仅仅用来显示数据和

处理与用户的交互，而无须处理数据的存储和维护等。作为中间层的应用程序服务器则用来协调和更新各客户应用程序的请求，同时处理所有定义数据集和与远程数据库服务器交互的细节。

多层应用模式的优点主要包括以下几个方面：

（1）共享中间层。在一个共享的中间层封装了商业规则。不同的客户程序可以共享同一个中间层，而不必由每个客户程序单独实现商业规则；

（2）瘦客户应用。客户程序可以做得很“瘦”。因为很多复杂的工作由应用服务器代劳了，客户程序只需要关注用户界面本身。“瘦”客户程序更容易发布、安装、配置和维护；

（3）分布数据处理。把一个应用程序分布在几个机器上运行，可以提高应用程序的运行效率，通过冗余配置还可以保证不会因为局部故障导致整个应用程序崩溃；

（4）安全性好。可以把一些敏感的功能放在有严密防护措施的层上，从而不至于使用户界面变得复杂。Delphi 4 中的 CORBA 或 MTS 支持较复杂的安全机制。



认识 MIDAS 技术



遇到难题

在上一节中已经了解了多层应用的一些特点和优点，那么 MIDAS 到底是一门什么样的技术呢？



钥匙在此

1. 基于 MIDAS 技术的多层应用概述

基于 MIDAS 技术的多层应用的运行过程如下：

（1）用户首先要启动客户程序，此时客户程序将试图连接应用服务器（这可以在运行或设计时指定），如果应用服务器还没有运行，客户程序将激活应用服务器。客户程序将从应用服务器获得一个 IProvider 接口；

（2）客户程序向应用服务器请求数据。如果 TClientDataSet 的 FetchOnDemand 属性被设为 True，那么客户程序会根据需要自动检索附加的数据包，如 Blob 字段的值或嵌套表的内容。否则，客户程序需要调用 GetNextPacket 才能获得这些附加的数据包；

（3）应用服务器收到客户程序的请求后，就从远程数据库服务器获得数据（如果必要的话，先要建立一个数据库连接），并打包返回给客户程序；

（4）客户程序收到数据包后把包打开，然后显示给用户；

（5）用户对数据进行编辑修改（如删除、添加或者修改），这些修改将被客户端程序存储在修改日志（change log）中；

（6）客户端程序向应用服务器申请更新数据，应用服务器一般会响应用户的操作。为了更新数据，客户端应用程序会将它的修改日志（change log）进行打包并发送给应用服务器。

（7）应用服务器收到客户程序的申请后，就向远程数据库服务器以事务方式更新数据。如果出错，应用服务器会试图修改当前的数据，或者存储不能更改的记录。这种提交记录以及存储出错的记录的方法称之为“分析”（resolving）；

（8）当应用服务器完成了这个“分析”过程（the resolving process）后，就把出错的记录返回给客户程序去核对。

（9）客户程序核对并修改了数据后，便可以放弃此次更新，也可以继续此次更新；

（10）客户程序从服务器上刷新显示数据。

2. 客户应用程序结构

对于最终用户来说，多层体系结构中的客户程序与两层体系结构中的应用程序没有什么区别。在结构上，客户程序就好像一个基于文件的单层应用程序一样，仍然通过标准的数据控件与用户进行交互。

但与单层应用程序不同的是，多层体系结构中的客户程序是通过应用服务器提供的接口来获得数据和申请更新数据。在大多数情况下，这个接口即为 *Iprovider* 接口。客户端应用程序是通过一个连接构件获得这个接口的。关于“*Iprovider* 接口”以及相应的连接构件将在后文中给以介绍。



当使用 MTS 的时候，可以不使用 IProvider 接口。不使用 IProvider 接口的好处是，可以充分发挥 MTS 在处理事务方面的特长。

连接构件用来建立客户应用程序和应用服务器之间的连接关系。在客户程序中，MIDAS 连接构件扮演着极其重要的角色。不同的 MIDAS 连接构件使用不同的通信协议。这些连接构件总结在表 20-1 中。

表 20-1　　连接构件

构　件	协　议
TDCOMConnection	DCOM
TSocketConnection	Windows sockets （TCP/IP）
TOLEnterpriseConnection OLEnterprise	（RPCs）
TCorbaConnection	CORBA （IIOP）



TRemoteServer 和 TMIDASConnection 是为了和 Delphi 3 的代码兼容而保留的。

一旦连接建立，连接构件就可以通过应用服务器的远程数据模块（remote data module）的 *IDataBroker* 接口为在客户端的远程数据模块中的数据集获得 *Iprovider* 接口。

3. 应用服务程序结构

应用服务器拥有一个远程数据模块，它提供了 IDataBroker 接口。当客户程序与应用服务器建立了连接后，就通过 IDataBroker 接口来获得 IProvider 接口。

Delphi 4 和 Delphi5 支持三种类型的远程数据模块：

TremoteDataModule 这是一个支持双重接口的自动化服务器，这种类型的远程数据模块适合于 DCOM、TCP/IP 或 OLEnterprise 方式。

TMTSDataModule 这也是一个支持双重接口的自动化服务器，用这种类型的远程数据模块创建的应用服务器是 Active Library 即动态链接库，适合于 DCOM、TCP/IP 或 OLEnterprise 方式。

TCorbaDataModule 这是 CORBA 服务器，适用于与 CORBA 客户通信。

上述三种远程数据模块都可以作为容器，但只能放置非可视的构件。另外，远程数据模块上一般要放一个或几个 TDataSetProvider 或 TProvider 构件来提供 IProvider 接口。远程数据模块也可以拥有 TDatabase 构件和 TSession 构件。*Iprovider* 一般能够完成以下操作：

- 从客户应用程序收到数据请求，然后从数据库服务器获得所请求的数据，接着打包事务数据，并且发送数据至客户端数据集。这种行为称之为“提供服务”（providing）；
- 从客户端获得修改的数据，并将这些修改向数据库服务器申请，然后存储不能修改的记录，并且返回给客户端不能修改的记录。这种行为称之为“分析”（resolving）。



在基于 BDE 数据集的应用服务器中，如果不提供 provider 构件的话，就应该动态产生一个 provider。如果增加了 provider 构件的话，你便可以获得更大的控制，并且可以支持其他类型的数据集。

通常 provider 使用基于 BDE 的数据集，正如你可以在两层应用中可以看到的那样。如果需要的话，你可以增加一个 database 和 session 构件，正如基于 BDE 的两层结构中的那样。

（1）使用 MTS。MTS 是 Microsoft Transaction Server 的简称，是 Microsoft 为分布式环境下进行事务处理所设计的服务接口。使用 TMTSDataModule 类型的远程数据模块的优势是：

① 安全机制　MTS 为应用服务器提供了基于角色的安全机制。每个客户都扮演着一种角色，这种角色决定他们能否访问远程数据模块的接口。TMTSDataModule 有一个函数叫 IsCallerInRole，可以用来检查客户的角色，然后有条件地开放该角色所允许的功能。

② 数据库处理缓冲池　MTS 提供了缓冲池的功能，并能把与数据库的连接放到池中，当一个客户不再需要连接时，另一个客户可以继续使用它。这样，应用服务器不必再次登录到远程数据库服务器进行连接。可能有的读者会想到，这个功能非常类似于 TDatabase 构件的 KeepConnection 属性。不过，要注意的是，如果用了 TDatabase 构件的话，KeepConnection 属性最好设为 False。

③ MTS 的事务功能 .MTS 提供了强大的事务处理能力，它的“两阶段提交”技术使得应用程序能够跨服务器处理事务。你可以将自己的 MTS 事务集成到应用服务器中以强化事务处理能力。MTS 可以跨过多个数据库，或者包括没有引入数据库中的功能。

④ 及时激活或无效　可以用 TMTSDataModule 类型的远程数据模块实现一个 MTS 服务器，这个 MTS 服务器能够根据需要自动地激活或无效，换句话说，只有当远程数据模块接收

到客户的连接请求时才创建模块的一个实例，这样能够最大程度地节省资源。

由此可见，MTS 服务器可以有两种工作方式，一是单实例方式，一个实例能够处理多个客户的请求，不过，如果客户较多的话，远程数据模块就成了瓶颈，制约着应用服务器的性能。二是多实例方式，每个客户请求连接时都会创建远程数据模块的一个实例，这样，几个客户就可以同时访问数据库而不需要排队。



在使用 MTS 时，直到远程数据模块被激活时数据库连接才能被打开。在应用程序运行前，要确保所有的数据集被关闭，并且数据库被断开。在程序中，必须加入代码来控制数据库的连接和断开。当数据模块激活时应连接数据库，当数据库模块被关闭时，数据库的连接应当断开。

（2）使用 IdataBroker 接口。应用服务器上的远程数据模块支持 IDataBroker 接口，因此当客户程序与应用服务器连接以后，客户程序上的 MIDAS 连接构件就开始查找 IDataBroker 接口。

IDataBroker 接口只定义了一个方法叫 GetProviderNames，客户程序的连接构件调用这个方法可以获得应用服务器上的一列 TdataSetProvider 构件和 TProvider 构件。客户端的 TClientDataSet 的 ProviderName 属性可以指定其中一个 TDataSetProvider 或 TProvider 构件。



当客户程序通过 IDataBroker 接口的 GetProviderNames 方法以及 TClientDataSet 的 ProviderName 属性指定了应用服务器上的一个 TDataSetProvider 或 TProvider 构件后，只要客户还在引用 IProvider 接口，远程数据模块的状态就应该保持，这与 MTS 的许多特点是有冲突的，也会与单实例的 CORBA 服务器发生冲突。

（3）使用 Iprovider 接口。客户程序与应用服务器之间通过 IProvider 接口交换数据，不过，大部分客户程序并不直接使用 IProvider 接口，而是通过 TClientDataSet 的属性和方法间接地使用 IProvider 接口。不过，也可以通过 Provider 属性获得 IProvider 接口，然后直接访问 IProvider 接口。

下面这个表列出了 IProvider 接口的属性和方法，同时列出了 TProvider 构件和 TClientDataSet 构件中与之对应的属性和方法。

表 20-2　　Iprovider 接口成员

IProvider	TProvider	TClientDataset
ApplyUpdates	ApplyUpdates	ApplyUpdates
Constraints 属性	Constraints	客户程序只能通过 IProvider 接口访问这个属性
Data	Data	Data
DataRequest	DataRequest	客户程序只能通过 IProvider 接口访问这个方法
Get_Constraints	Constraints 属性	客户程序只能通过 IProvider 接口访问这个方法

续表

IProvider	TProvider	TClientDataset
Get_Data	Get_Data	用于实现 Data 属性
GetMetaData	GetRecords（Count = 0）	内部使用
GetRecords	GetRecords	用于 GetNextPacket
Reset	Reset	内部使用
Set_Constraints	Constraints 属性	客户程序只能通过 IProvider 接口访问这个属性
SetParams	SetParams	用于 Params 属性



IProvider 接口的许多属性和方法依赖于远程数据模块的状态信息，正因为如此，在使用 CORBA 或 MTS 的应用程序中一般不要用 IProvider 接口。

4. 选择连接协议

在客户程序与应用服务器之间，Delphi 4 和 Delphi 5 提供了四种不同类型的连接方式或者说通信协议，即 DCOM、TCP/IP、OLEnterprise 和 CORBA。这些不同的连接方式都各有利弊，到底选择哪种连接方式，取决于客户的数量、客户的分布情况以及发布应用程序的方式。

（1）使用 DCOM 连接。DCOM 是一种最直接的连接方式，它不需要专门的软件支持。不过，Windows 95 不支持 DCOM，除非安装了 DCOM95 程序（可以到微软服务站点去下载），但 Windows 98 在安装时已经自带了 DCOM。

可以使用分布式组件对象模型 （DCOM） 来集成网络中的分布式应用程序。分布式应用程序由多个进程组成，来共同完成一项任务。可以使用 DCOM 配置工具来配置 32 位 COM 和 DCOM 应用程序。

在使用 DCOM 应用程序之前，必须先设置应用程序的属性，诸如安全性和位置等。在运行客户应用程序的计算机上，应指定将被访问或将被启动的服务器应用程序的位置。而对于应用服务程序，则应指定有权访问或启动该应用程序的用户帐号，以及将运行该应用程序的用户帐号。具体如何配置，请运行 Dcomcnfg.exe，并参照相应的帮助。

如果需要使用 MTS 安全服务时，最好使用 DCOM 连接方式。MTS 的安全服务是基于角色的，当一个客户通过 DCOM 访问 MTS 时，DCOM 会告诉 MTS 有关客户的信息，MTS 据此来决定客户的角色。如果用其他连接方式，需要有专门的运行期软件支持，客户的调用首先被传递给这些运行期软件而不是 MTS，于是 MTS 就不能尽快指派角色。

（2）使用 TCP/IP 连接。TCP/IP 连接方式的适合范围非常广泛。例如，如果客户程序要以 ActiveForm 的形式发布在 Web 上，最好采用 TCP/IP 连接方式。因为您无法确定下载 ActiveForm 的计算机是否支持 DCOM，而支持 TCP/IP 的环境是很普遍的。

要使用 TCP/IP 连接方式，则在应用服务器端必须运行一个专门的运行期软件 ScktSrver.exe 或 ScktSrvc.exe，其中，ScktSrvc.exe 只适合于 Windows NT，可以作为一个服务在后台运行。与 DCOM 连接方式不同的是，客户的请求首先传递给 ScktSrver.exe 或 ScktSrvc.exe，然后再创建远程数据模块的实例，而不是由客户的调用直接创建远程数据模块的实例。客户程序上的 MIDAS 连接构件通过 IProvider 接口与 ScktSrvr.exe or ScktSrvc.exe

通信。

不过，客户程序很有可能在没有正常释放对 IProvider 接口的引用之前出现异常，而 TCP/IP 连接方式无法检测到这种情况，更无法通知应用服务器，因此，有可能造成应用服务器上的资源被占用后得不到释放的后果。

（3）使用 OLEnterprise 连接。如果要在应用服务器端使用 Business Object Broker 时，就要使用 OLEnterprise 连接方式。此时，应用服务器端和客户端都要安装 OLEnterprise 运行期软件。

（4）使用 CORBA 连接。

Delphi 4 和 Delphi 5 是目前能支持 CORBA 的开发工具之一。基于 CORBA 的客户程序和应用服务器可以与其他基于 CORBA 的应用程序无缝对接。要使用 CORBA 连接方式，需要 ORB 的支持，它提供了类似于 Business Object Broker 的功能。



跟我来

建立多层数据库应用的一般步骤如下：

1. 创建应用服务器

2. 注册应用服务器

（1）如果应用服务器使用了 DCOM、TCP/IP 或者 OLEnterprise 通信协议时，应用服务器应当作为一个自动服务器必须注册，就像 ActiveX 或者 COM server 一样。

（2）如果应用服务器使用了 MTS，则应用服务器必须是个 Active Library 而不是一个.exe。因为所有的 COM 调用都是通过 MTS 代理，所以不必注册应用服务器。相反，你应当将其安装至 MTS 中。

（3）如果应用服务器使用了 CORBA 通信协议，注册是可选项，但是推荐最好使用注册。如果允许客户应用程序动态绑定到你的接口上，则在接口库中必须安装服务器接口。另外，如果在应用服务器未运行时允许客户应用程序自动启动应用服务器，则必须以 OAD（Object Activation Daemon）注册。

3. 创建客户端应用程序

其中创建顺序比较重要。要创建一个多层 Client/Server 应用程序，首先要创建应用服务器，然后注册或安装应用服务器，只有应用服务器已注册并且正在运行的情况下，才能创建客户程序。对于客户程序来说，既可以在设计期连接应用服务器，也可以在运行期连接应用服务器。



如果你没有在应用服务器的同一个系统中创建客户应用程序，则需要在客户系统中注册和安装应用服务器。在设计的时候，可以在 Delphi 开发环境的对象观测器中选择 server names 和 provider names。



创建应用服务器



遇到难题

这是建立多层数据库应用的第一步。



钥匙在此

创建一个应用服务器与创建一个两层的数据库应用程序有些相似，主要的区别是，应用服务器需要提供 IProvider 接口，这一般是通过 TDataSetProvider 或 TProvider 构件来提供的，也可以通过数据集构件如 TTable 的 Provider 属性提供。



跟我来

为了创建应用服务器，首先必须建立一个新项目，并存储它，然后按照以下步骤进行：

（1）新建一个新远程数据模块并放于该项目中。从主菜单中选择“File|New”，然后在“Object Repository”对话框中选择“Multitier”页，如图 20-1 所示，并选择：

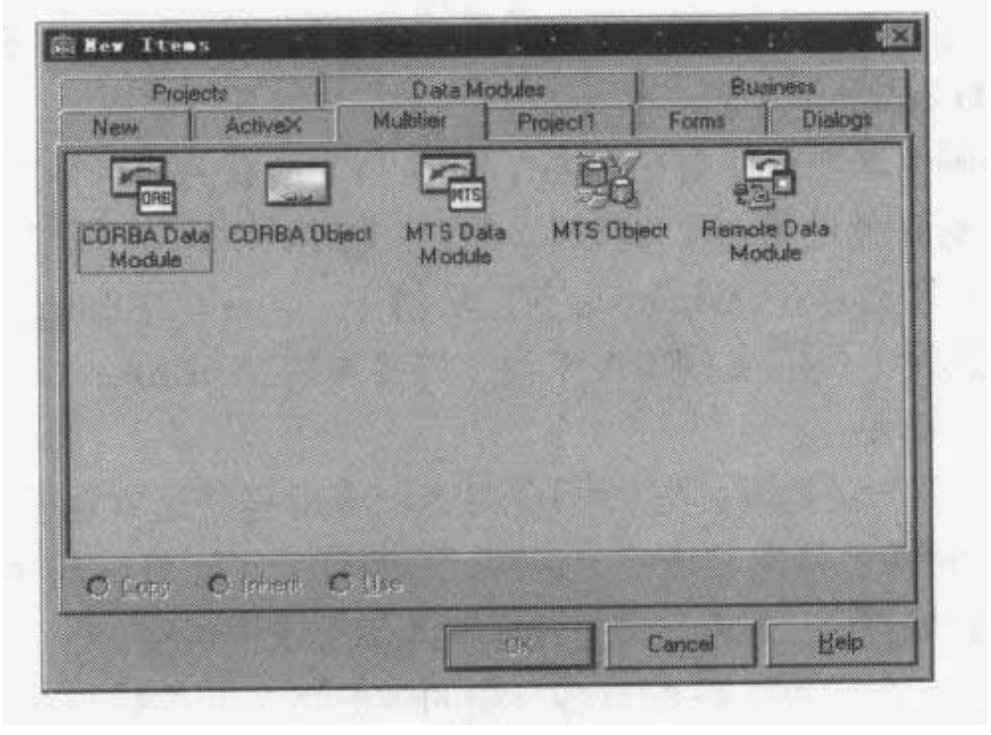


图 20-1　选择一个远程数据模块

① 选择一个远程数据模块。 如果要创建一个 COM 自动化服务器，允许客户通过 DCOM、TCP/IP 以及 OLEnterprise 等方式访问此服务器时，选择远程数据模块。

② 选择 MTS 数据模块。如果要创建一个允许客户通过 MTS 访问的 Active Library 时，选择 MTS 数据模块。

③ 选择 CORBA 数据模块。如果要创建 CORBA 服务器时，选择 CORBA 数据模块。



远程数据模块比简单的数据模块能发挥更大的作用。CORBA 数据模块充当的是 CORBA 服务器的角色，其他数据模块则是 COM 自动对象。

（2）把一个数据集构件如 TTable、TQuery 或 TstoredProc 等放到远程数据模块上，并进行有关设置，使得它们能访问远程的 SQL 数据库。尽量不要把 TDatabase 构件放到远程数据模块上，因为这可能引起名称冲突。如果实在要用 TDatabase 构件来连接 SQL 数据库，建议把 TDatabase 构件放到另一个数据模块上，然后引用这个数据模块的单元文件。

（3）把 TDataSetProvider 或 TProvider 构件放到远程数据模块上。对于每一个数据集构件，需要有--个 TDataSetProvider 或 TProvider 构件与之对应。然后，用鼠标右键单击 TDataSetProvider 或 TProvider 构件，在弹出的菜单中选择“ExportFrom <Name> in Data Module”命令，这是为了引出 IProvider 接口，在类型库中注册。

（4）设置由 TDataSetProvider 或 TProvider 构件的 DataSet 属性所指定要访问的数据库，实际上就是第二步所放的数据集构件。

（5）编写代码以实现事件、共享的商业规则以及共享的数据有效性和安全性。

（6）保存、编译、注册或安装应用服务器。

① 如果使用 DCOM、TCP/IP 或者 OLEnterprise 作为通信协议，应用服务器就好像一个自动化服务器一样，必须像 ActiveX 或 COM 服务器那样进行注册。

② 如果使用 MTS，应用服务器就是 DLL 而不是 EXE，这时候不需要注册应用服务器，而是需要把这个 DLL 作为 MTS 对象安装到 MTS 包中。

③ 如果使用 CORBA，可以不注册，但最好注册。如果要使客户程序对服务器接口的调用是在运行期进行动态确定，就要在接口库（Interface Repository）中安装服务器的接口。如果要使客户程序能自动激活应用服务器(如果还没有运行的话)，应用服务器就必须用 OAD（Object Activation Daemon）注册。

（7）如果应用服务器没有使用 DCOM，就必须安装有关的运行期软件，因为其他连接方式需要这些运行期软件的支持。例如，对于 TCP/IP 来说，需要安装 ScktSrvr.exe 或 ScktSrvc.exe，后者只能运行在 Windows NT 环境下；对于 OLEnterprise 来说，需要安装 OLEnterprise 运行期版本；对于 CORBA 来说，需要安装 VisiBroker ORB。

1. 设置远程数据模块

在设置和运行应用服务器时，它并不会同客户应用程序建立连接。而是由客户应用程序通过连接组件与应用服务器建立连接，通常，一旦建立了连接，则会选择一个 provider 接口。所有这些都是自动进行，不必写代码管理输入请求或提供接口。

一旦建立了连接，在应用服务器与客户端的数据集之间的接口对开发人员和用户都是自动和透明的。连接由客户应用程序来维护。

当你创建了远程数据模块后，你必须提供确定的信息来指定如何响应客户端的请求。

这些信息随着远程数据模块的类型不同而不同。

应用服务器的关键部件是远程数据模块。Delphi 4 支持三种类型的远程数据模块，分别是 TRemoteDataModule、TMTSDataModule 和 TCorbaDataModule。

（1）设置 TremoteDataModule。要加入一个 TRemoteDataModule 类型的远程数据模块，首先要使用“File”菜单上的“New”命令，然后，选取“Multitier”页，双击“Remote Data Module”图标，便会弹出“Remote Data Module Wizard”对话框，如图 20-2 所示。

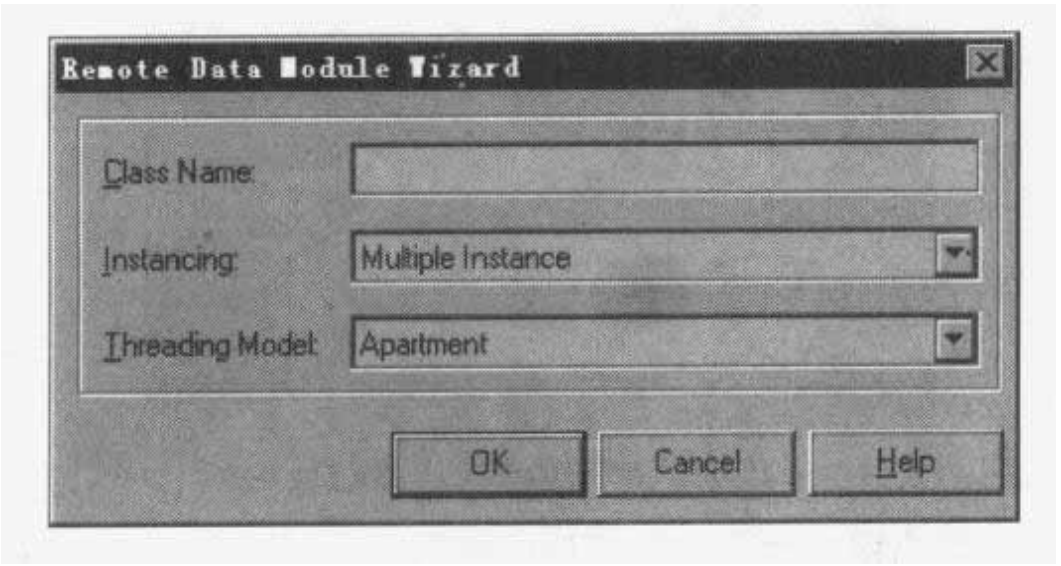


图 20-2　Remote Data Module 对话框

在“Class Name”框内键入远程数据模块的类名，不必以 T 打头。Delphi 4 将根据此名生成一个 TRemoteDataModule 的派生类和有关接口。例如，假如在“Class Name”框内键入“MyDataServer”，远程数据模块的类名就是 TMyDataServer，它所实现的接口就叫 IMyDataServer，其祖先接口是 IDataBroker。

提示：你可以在接口中增加自己的属性和方法。

如果你要产生一个 DLL（Active Library），就必须在远程数据模块向导中指定线程的模式。可以在“Threading Model”框内选择一种线程模式，可以选“Single-threaded”、“Apartment-threaded”、“Free-threaded”或者“Both”。

① 如果选择 Single-threaded，COM 会确保一次服务一个客户应用程序的请求。而不必担心客户程序间的互相干扰。

② 如果选择 Apartment-threaded，COM 会确保一个远程数据模块实例一次响应一个请求。当在 Apartment-threaded 库中写代码时，如果使用全程变量或者对象而没有置于远程数据模块中时，则必须防止线程冲突。

③ 如果选择 Free-threaded，应用程序在几个线程中可以接收同时发生的客户请求，此时必须确保线程的安全。因为多个客户可以同时访问你的远程数据模块，你必须确保实例数据及全程变量的安全。这种模式并不推荐，因为没有内建线程支持 *IProvider* 接口。

④ 如果选择 Both，就如同选择 Free-threaded 一样，并带有一个异常。

如果你产生一个.exe，必须指明使用的实例类型。在“Instancing”框内选择是否根据客户的请求生成远程数据模块的多个实例，可以选“Single instance”或“Multiple instance”。

⑤ 如果选择了“Single instance”，每一个客户程序将启动一个自己可执行的应用实例。这一远程数据模块实例将只为这一客户连接服务。

⑥ 如果选择了“Multiple instance”，单个应用实例（线程）将为客户程序初始化所有的远程数据模块，此时一个远程数据模块实例将服务于一个客户连接。但所有这些远程数据模

块都共享一个线程空间。

（2）设置 TMTSDataModule。要加入一个 TMTSDataModule 类型的远程数据模块，首先需要使用“File”菜单上的“New”命令，然后选择“Multitier”页，双击“MTS Data Module”图标，此时会弹出“MTSData Module Wizard”对话框，如图 20-3 所示。

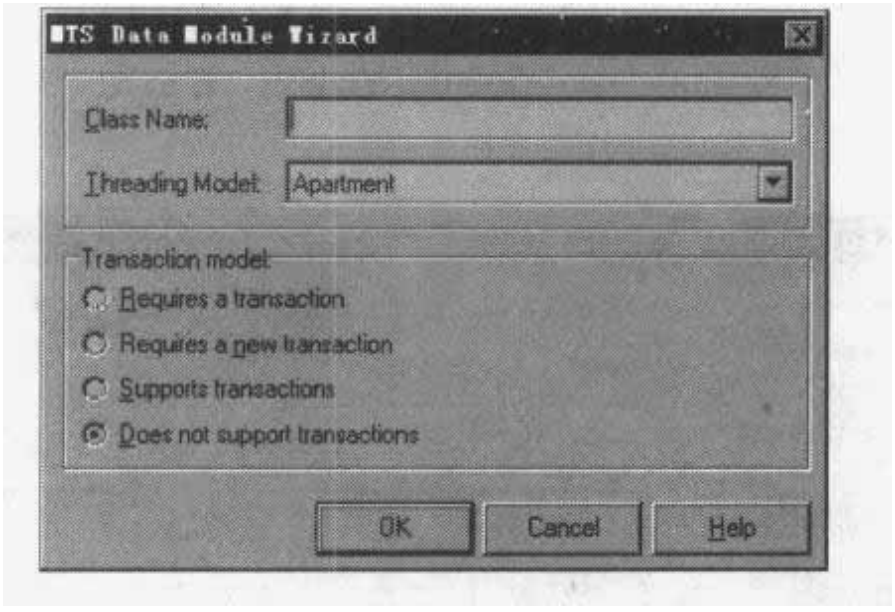


图 20-3　MTS Data Module 对话框

在“Class Name”框内键入远程数据模块的类名，不必以 T 打头。Delphi 4 和 Delphi 5 将以此名生成一个 TMTSDataModule 的派生类和有关接口。

例如，假设在“Class Name”框内键入“MyDataServer”， 远程数据模块的类名就是 TMyDataServer，它所实现的接口叫 IMyDataServer，其祖先接口是 IDataBroker。

对于 TMTSDataModule 类型的远程数据模块来说，必须在“ThreadingModel”框内选择一种线程模式。可以选“Single”、“Apartment”或者“Both”。



在 MTS 中的 Apartment 和 Free 模式 与 DCOM 下的响应模式是不一样的。具体的信息可以参见 Delphi 的在线帮助。

在“Transaction Attributes”框内选择事务属性：如选择“Requires a transaction”，每当客户访问远程数据模块的接口时，都与当前的事务是相关的。客户不可能在事务中再申请一个新的事务。

① 如选择“Requires a new transaction”，每当客户访问远程数据模块的接口时，都自动开始一个新的事务。

② 如选择“Supports transactions”，远程数据模块可以用在事务的环境中，客户访问远程数据模块的接口时必须申请一个新的事务。

③ 如选择“Does not support transactions”，远程数据模块不能用在事务的环境中。



MTS 对象只能加入到 ActiveX 项目中。

（3）设置 TCORBADataModule

要加入一个 TCorbaDataModule 类型的远程数据模块，需要首先使用“File”菜单上的“New”命令，然后选取“Multitier”页，双击“CORBA Data Module”图标，此时会弹出“CORBA Data Module Wizard”对话框，如图 20-4 所示。

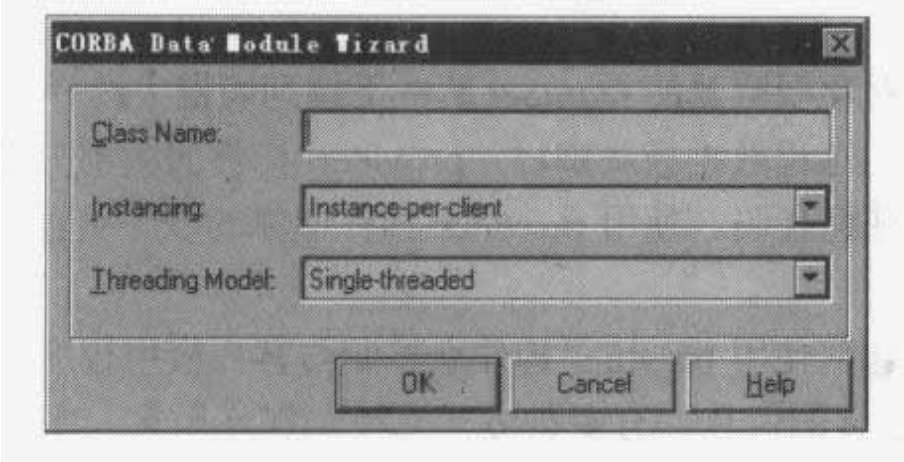


图 20-4　CORBA Data Module 对话框

在“Class Name”框内键入远程数据模块的类名，不必以 T 打头。Delphi 4 和 Delphi 5 将以此名生成一个 TCorbaDataModule 的派生类和成有关接口。例如，假设在“Class Name”框内键入“MyDataServer”， 远程数据模块的类名就是 TMyDataServer，它所实现的接口叫 IMyDataServer，其祖先接口是 IDataBroker。

在“Instancing”框内指定应用服务器创建数据模块的实例的方式，可以选“Shared Instance”或者“Instance-Per-Client”。

① 如果选“Shared Instance”，应用服务器只创建数据模块的一个实例来处理所有客户的请求，因此，远程数据模块必须与状态无关，换句话说，就是不能使用 IProvider 接口。

② 如果选“Instance-Per-Client”，每当一个客户试图连接时，远程数据模块都会生成一个实例。只要客户与应用服务器的连接没有断开，远程数据模块的实例就一直存在。这种模式下，允许使用 IProvider 接口。唯一要考虑的问题是，客户程序有可能意外终止，会导致没有正常地断开与应用服务器的连接。应用服务器为了避免不必要的资源浪费，可以定期地检查客户是否正在运行，如没有，就手工把远程数据模块的实例删掉。



不像 COM 服务器的实例，模式决定了线程实例的数目，在 CORBA 中，则是实例决定对象实例的数目。它们都是在执行服务器单一实例中创建的。

另外，在“Threading Model”框内选择一种线程模式，可以选“Single-threaded”或“Multi-threaded”。

2. 为应用服务器创建数据 provider

远程数据模块上往往要放一个或几个 TDataSetProvider 或 TProvider 构件，用于提供 IProvider 接口。provider 构件负责 *IProvider* 接口，并负责打包送至客户端的数据以及申请更改。Providers 与负责处理数据分解至数据库服务器（或者数据集）详情的 resolver 构件协同工作。



也可以不直接使用 TDataSetProvider 或 TProvider 构件，而是由数据集构件如 TTable、TQuery 或 TStoredProc 的 Provider 属性间接地提供 IProvider 接口。显式地使用 TDataSetProvider 或 TProvider 构件的好处是，可以直接控制数据包中包含信息的内容以及应用服务器怎样响应客户请求的。

如果直接使用了 TDataSetProvider 或 TProvider 构件，必须设置他们的 DataSet 属性来指

定要访问的数据集。

在设计时，必须从对象观测器的 *DataSet* 属性下拉列表框中选择可用的数据集。

不像创建一个无度的远程数据模块那样，provider 构件的大部分工作都是自动产生的，你不必在 provider 中写任何代码。使用 provider 构件使你的程序更直接地控制数据打包以及如何响应客户请求。

下面的主题描述了如何使用 provider 构件来控制与客户应用程序的交互。

（1）控制数据包包含的信息。有多种方式来控制在送至客户端和从客户端获取的数据包中包括哪些信息。这些方式有：

① 指定有哪些字段包含在数据包中；

② 设置影响数据包的选项；

③ 增加定制信息至数据包中。

指定数据包中的字段

要控制哪些字段包含到数据包中，首先要创建永久字段。之后，只有永久字段才能加入到数据包中。在服务器上数值动态变化的字段（例如计算字段和查找字段），也能包括进去，但这些字段传递到客户端后就会变成只读。

由于客户程序很有可能要编辑和修改数据，并且要把编辑和修改后的数据申请更新到应用服务器上，因此，您创建的永久字段的数量不能太少，以避免出现重复的记录。否则，当这些更改被申请时，就无法确定哪一个记录将被修改。如果实在不想使客户程序看到某个字段，而如果没有这个字段的话又很有可能出现上述错误，这时候您可以让这个字段（TField 对象）的 ProviderFlags 属性包含 pfHidden 元素，表示这个字段虽然加入到数据包中，但却是隐含的，客户看不到它。



如果使用 TQuery 作为应用服务器上的数据集构件，那么 SQL 语句应当选择足够多的字段，即使客户程序并不需要这么多字段。否则，就有可能出现上述错误。

设置 Options 属性

provider 构件的 Options 属性是一个集合，用于设置有关打包和传递的选项，确定 BLOBs 或者嵌套的明细表什么时候被发送，字段属性是否显示等等。表 20-3 列出了在属性中可能出现的值。

表 20-3　　Provider 选项

值	意 义
poFetchBlobsOnDemand	表示 BLOB 字段一般不放到包中，除非客户端的 TClientDataSet 构件的 FetchOnDemand 属性设为 True 或者显式地调用 FetchBlobs
poFetchDetailsOnDemand	表示嵌套表中的字段不放到包中，除非客户端的 TClientDataSet 构件的 FetchOnDemand 属性设为 True 或者显式地调用 FetchDetails
poIncFieldProps	表示把字段的属性也放到包中，包括 Alignment、MinValue、DisplayLabel、DisplayWidth、Visible、DisplayFormat、MaxValue、EditFormat、Currency、EditMask、DisplayValues 等属性
poCascadeDeletes	当父表中的某条记录被删除时就把子表中的相应记录也删除
poCascadeUpdates	当父表的关键字段的值变化时自动更新子表的记录
poReadOnly	表示不允许“瘦”客户向 TDataSetProvider 申请更新数据

在数据包中加入自定义的信息

当客户端通过 IProvider 接口调用 DataRequest 函数请求数据时，将在应用服务器端触发 OnGetDataSetPropertiesevent 事件，这时，应用服务器就有机会在数据包中加入一些自定义的信息。这些信息以一个 OleVariant 类型变量对信息进行编码，并存储在指定的名字中。客户程序中的 TclientDataset 可以使用 *GetOptionalParam* 方法来获得信息。你也可以指定被包含在 delta 包中的信息，这些包是 TclientDataset 在修改记录时发送的。在这种情形下，客户程序永远不会明了这些信息，但服务器会发送一个 round-trip 消息给它。

当加入定制信息到 *OnGetDataSetProperties* 事件中时，每一个独立的参数（有时称之为选项参数）都是由一个可变数组来指定的。OnGetDataSetPropertiesevent 事件是这样声明的：

TGetDSProps = Procedure（Sender: TObject; DataSet: TDataSet; out Properties:OleVariant）；

其中， Properties 参数是一个可变类型的数组，用于指定要加入的信息。Properties 参数的每个元素由三部分组成：名称（字符串）、值（变体类型）和一个布尔标志（布尔类型）。这个布尔标志确定了 TclientDataset 在修改记录时信息是否被包含在 delta 包中。多个参数可以通过产生变体类型数组来加入。例如，下列 *OnGetDataSetProperties* 事件发送了两个值，即数据提供的时间和源数据集的总记录数。当客户申请更改时仅仅只有提供数据的时间信息被返回。程序示例如下：

```
Procedure TAppServer.Provider1GetDataSetProperties（Sender: TObject; DataSet: TDataSet;
out Properties:OleVariant）；
    Begin
    Properties := VarArrayCreate（[0， 1]， varVariant）；
    Properties[0] := VarArrayOf（['TimeProvided'， Now， True]）；
    Properties[1] := VarArrayOf（['TableSize'， DataSet.RecordCount， False]）；
    End;
```

上面这个程序中，加入了两个自定义的信息，一个叫 TimeProvided，它的值是当前的日期和时间，True 表示这个信息可以由客户端返回给应用服务器。另一个信息叫 TableSize，它的值是数据集的记录数，False 表示这个信息不可以由客户端返回给应用服务器。

以后，当客户端申请更新数据时，提供原来数据的时间可以在 TDataSetProvider 的 OnUpdateData 事件中读出。程序示例如下：

```
Procedure TAppServer.Provider1UpdateData（Sender: TObject; DataSet: TClientDataSet）；
varWhenProvided: TDateTime;
Begin
    WhenProvided := DataSet.GetOptionalParam（'TimeProvided'）；
    ...
End;
```

（2）响应客户的数据请求。在大多数的多层应用程序中，客户对数据的请求是被自动处理和响应的。客户端通过调用 *GetRecords*（通过 *IProvider* 接口或者通过自定义接口）方法来请求打包数据。Provider 运输数据，产生数据包并送至客户端自动响应。

应用服务器在把数据包传递给客户之前，还有机会对其中的数据进行编辑加工，例如，可以对其中敏感的数据加密，或者根据某种条件删掉一些记录。

为了在送至客户端之前修改数据，可以在 *OnGetData* 事件写入修改代码。数据包是以客户数据集 TClientDataSet 的形式作为参数来提供的。该事件声明如下：

```
type TProviderDataEvent = procedure（Sender: TObject; DataSet: TClientDataSet） of object;
property OnGetData: TProviderDataEvent;
```

因此，使用客户数据集 TclientDataSet 的一些方法，便可实现在送至客户端之前进行数据修改。如下列程序：

```
procedure TDBClientTest.ProviderGetData （Sender: TObject; DataSet: TClientDataSet）;
begin
  with DataSet do
  begin
    while not EOF do
    begin
      Edit;
      SensitiveData.AsString := DoEncrypt（SensitiveData.AsString）;//进行加密处理
      Post;
      Next;
    end;
  end;
end;
```

（3）响应客户修改请求。客户程序通过调用 *ApplyUpdates* 方法（通过 *IProvider* 接口或者通过自定义接口） 向应用服务器申请更新数据。当应用服务器上的 TDataSetProvider 或 TProvider 构件收到客户的更新请求后，就会触发 *OnUpdateData* 事件。在这个事件中，你可以加入代码对 *Delta* 数据包进行修改。这样，在写入应用服务器的数据集在申请更改之前有机会编辑数据包（Delta 属性）。在 *OnUpdateData* 事件后，TDataSetProvider 或 TProvider 构件就会使用关联的 resolver 构件把数据更新送到远程服务器上。

Resolver 构件对数据的更新是一条记录一条记录进行的。在 Resolver 申请更改每一条记录之前将触发 provider 的 *BeforeUpdateRecord* 事件，从而使您还有机会对数据进行筛选和更改。如果出现错误，Resolver 就会触发 provider 的 *OnUpdateError* 事件来解决。发生错误的原因通常是数据违反了服务器的约束，或者另一个客户程序正好在前一个客户已经申请更新的时候修改了该记录。

这些修改错误既可以由应用服务器来处理，也可以回传给客户处理。应用服务器处理那些不需用户交互来解决的问题。当应用服务器不能解决这些问题时，它会临时储存这些错误记录的拷贝。当记录修改全部完成了，应用服务器就会将错误发生的个数返回给客户的 TclientDataset，并且将这些错误记录放入结果数据包中传给客户端。有些错误可能需要用户的介入，这时就要客户端进行处理。



存储过程构件是不能按照这种机理来更改数据的。为了向存储过程申请更改，你必须按照以下步骤来实现：

① 首先在 BeforeUpdateRecord 事件中明确加入代码说明要申请更改数据。典型的，需

要使用其他的存储过程构件来对数据进行编辑、插入和删除等。

② 当通过 SQL 来申请数据更改时应提供一个 Resolver 可用的表名。这种方式仅仅当存储过程从单一表中取数据时才有效。通过数据包的'Table_Name'属性来指定表名，并指定该属性包含在 Delta 数据包中。

（4）在更新数据库之前编辑 Delta 数据包。在 provider 向数据库申请更改数据之前，它会产生一个 *OnUpdateData* 事件。这个事件将得到一个 *Delta* 数据包的一份拷贝，这个拷贝是作为参数传过来的。这个参数就是 TClientDataSet 。OnUpdateData 事件是这样声明的：

TProviderDataEvent = Procedure（Sender: TObject; DataSet: TClientDataSet） of object;

其中，DataSet 参数代表客户程序上的 TClientDataSet 构件，这样就可以访问 Delta 属性得到当前要更新的数据包。

在 *OnUpdateData* 事件中，你可以使用 Tclientdataset 的方法或者属性在数据送入给应用服务器上的数据集之前修改 Delta 数据包。一个非常有用的属性是 *UpdateStatus* 属性。这个属性将指明在 delta 包中对当前记录修改的类型是什么。这些类型值如表 20-4 所示。

表 20-4　　UpdateStatus 值

值	意　义
usUnmodified	记录内容没有被改变
usModified	记录内容已经改变
usInserted	记录被插入
usDeleted	记录被删除

例如，下例 *OnUpdateData* 事件将当前的日期插入到每条新记录中，这些新记录是被插入到数据库中的。

procedure TMyDataModule1.Provider1UpdateData（Sender: TObject; DataSet: TclientDataSet）;

```
begin
        with DataSet do
begin
    First;
while not Eof do
begin
    if UpdateStatus = usInserted then
begin
    Edit;
    FieldByName（'DateCreated'）.AsDateTime := Date;
    Post;
end;
Next;
end;
end;
```

```
end;
```

（5）确定更改记录和字段。在你的 provider *OnUpdateData* 事件中还可以确定数据包中的记录在数据库中怎样进行更改。换句话说，哪些字段和记录将更新到服务器上。

缺省情况下，应用服务器用自动生成的 SQL UPDATE、INSERT 或 DELETE 语句来把 Delta 数据包中的记录写到远程服务器中，例如：

```
UPDATE EMPLOYEES
    Set EMPNO = 748，  NAME = 'Smith'，  TITLE = 'Programmer 1'，  DEPT = 52
WHERE
EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

除非另外指定，否则数据包中的所有字段都将出现在 UPDATE 子句和 WHERE 部分，换句话说，会更改记录所有的字段和用所有的字段去定位一条记录。不过，也可以有选择地排除一些字段，这就要用到 UpdateMode 属性，即更改模式属性。这个属性可以设为表 20-5 所列的值。

表 20-5　　UpdateMode 属性取值

值	描　　述
upWhereAll	所有字段都用来定位记录（在 Where 子句中）
upWhereChanged	只有关键字段和变化了的字段用来定位记录
upWhereOnly	只有关键字段用来定位记录

不过，UpdateMode 属性的控制还是有一定的局限性，你还可以有更多的控制。例如，在以前的 SQL 语句中，为了防止 EMPNO 字段被修改而将 EMPNO 字段不放入 UPDATE 子句中，而且不想让 TITLE 和 DEPT 字段出现在 WHERE 部分以避免修改冲突。为了指定哪些字段出现在哪些子句中，要用到 *ProviderFlags* 属性。此属性是一个集合，可以包含表 20-6 所列的值。

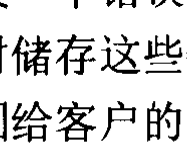
表 20-6　　ProviderFlags 属性取值

值	描　　述
pfInWhere	该字段将不出现在自动生成的 INSERT、DELETE 和 UPDATE 语句的 WHERE 部分
pfInUpdate	该字段将不出现在自动生成的 UPDATE 语句的 UPDATE 子句
pfInKey	这个字段将出现在因为更新失败而执行的 SELECT 语句的 WHERE 部分，SELECT 语句用于选择出错的记录的当前值
pfHidden	这个字段将用来定位字段，但对客户端是隐藏的

下面这个程序示例会把 EMPNO 字段排除在 UPDATE 子句之外，同时把 TITLE 字段和 DEPT 字段排除在 WHERE 部分之外。

```
procedure TMyDataModule1.Provider1UpdateData ( Sender: TObject; DataSet:
TClientDataSet）;
```

```
    begin
    with DataSet do
    begin
        FieldByName（'EMPNO'）.UpdateFlags := [ufInUpdate];
        FieldByName（'TITLE'）.UpdateFlags := [ufInWhere];
        FieldByName（'DEPT'）.UpdateFlags := [ufInWhere];
    end;
end;
```



即使你正在更改数据集而没有动态生成 SQL 时。你也可以使用 *UpdateFlags* 属性来确定数据如何被更改，这个属性仍可以确定哪些字段用于定位记录，哪些字段将被修改数值。

（6）过滤无效修改。在每一个更改申请之前，provider 会收到 *BeforeUpdateRecord* 事件，你可以利用这个事件在更改申请之前来编辑记录。同样的方式，你可以使用 *OnUpdateData* 事件来修改整个 delta 数据包。

另外，你可以使用这个事件对自己，申请的更改进行过滤或拒绝更改。在 *BeforeUpdateRecord* 事件中可以加入代码通知 resolver 一个更改已经处理了或者不该被申请。然后，resolver 就会跳过这条记录，也不会再增加错误的个数。例如，这个事件提供了一个机理，在向存储过程（该存储过程不会自动更改）申请更改时，或者一旦在事件中记录被修改允许 provider 跳过自动处理。

（7）在 provider 中解决更改错误。在应用服务器试图提交 delta 数据包中的一条记录而引发一个错误时，*OnUpdateError* 事件就会发生。当应用服务器不能解决这些问题时，它会临时储存这些错误记录的拷贝。当记录修改全部完成后，应用服务器就会将错误发生的个数返回给客户的 TclientDataset，并且将这些错误记录放入结果数据包中，并传给客户端。

这种机理，可以让你在应用服务器上解决一些问题，也可以留给客户端的用户进行交互方式修改错误。

OnUpdateError 事件声明如下：

```
type
  TResolverErrorEvent = procedure（Sender: TObject; DataSet: TClientDataSet;
    E: EUpdateError; UpdateKind: TUpdateKind; var Response: TResolverResponse） of
    object;
  property OnUpdateError: TResolverErrorEvent;
```

例如，当一个客户删除一条已被另一个用户删除了的记录，这时一条错误信息就会被返回给客户端。下面的例子演示了如何跳过这一错误而不返回给客户端。程序如下：

```
procedure TDBClientTest.ProviderUpdateError（DataSet: TClientDataSet;
  E: EUpdateError; UpdateKind: TUpdateKind; var Response: TResolverResponse）;
```

```
begin
  if （UpdateKind = ukDelete） and
     （E.ErrorCode = DBIERR_OPTRECLOCKFAILED） then
    Response := rrIgnore else
    Response := rrSkip;
end;
```

如果在该事件中能解决这些错误，可以设置 *Response* 参数以便让正确的记录被使用。

（8）响应客户产生事件。Provider 构件可以允许自己从客户端能直接访问 provider 调用的事件。这就是 *OnDataRequest* 事件。

OnDataRequest 事件是 provider 的特殊功能的一部分。这是允许你的客户端直接与应用服务器 providers 通信的一个简单的钩子（hook）。该事件声明如下：

```
type TDataRequestEvent = function（Sender: TObject; Input : OleVariant） : OleVariant of object;
property OnDataRequest: TDataRequestEvent;
```

这个事件句柄以一个 OleVariant 变量作为输入参数，并返回一个 OleVariant。通过使用 OleVariant，这使得容纳你与 provider.通信的任何信息就显得比较容易了。

（9）处理服务器约束。大多数数据库管理系统（RDBMS）都需要保证数据的完整性和一致性的约束。所谓约束，实际上就是预先指定一些规则，字段和记录的值以及表间字段的关系等必须符合这些规则。例如，大多数符合 SQL-92 的 RDBMS 都支持下列约束：

• NOT NULL	字段必须有值
• NOT NULL UNIQUE	字段必须有值而且不能与其他记录重复
• CHECK	字段的值必须在一个范围内
• CONSTRAINT	在表格级对字段的值进行检查
• PRIMARY KEY	指定一个或几个字段作为关键字段
• FOREIGN KEY	指定一个或几个字段引用其他表格



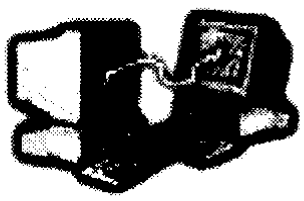
不是所有的数据库都支持上述约束，也有的服务器还支持其他约束。关于这方面的信息可以参考相应的数据库服务器的文档。

在你的多层数据库应用中，应当充分实现数据库服务器的约束而不必在客户程序或者应用服务器中加入代码重复这些约束。

Provider 的 *Constraints* 属性可以使你重复并使用数据库服务器的约束。只要把这个属性设为 True（默认），应用服务器就可以借用远程数据库服务器的约束，对客户程序传递过来的数据进行纠错。



有时对送至客户端的数据不需要使用约束。如果不想借用远程数据库服务器的约束，应当把 Constraints 属性设为 False。



创建客户端应用



遇到难题

这是建立多层数据库应用的第三步。



钥匙在此

一般情况下，建立多层体系中的客户应用程序与创建传统的两层体系中的客户程序有些类似。主要不同的地方体现在多层的客户应用程序使用了：

（1）与应用服务器连接的连接构件；

（2）一个或多个连接应用服务器数据 provider 的 *TclientDataSet* 构件。数据明了控件通过 DataSource 构件与 *TclientDataSet* 构件进行连接，而不是通过 DataSource 构件与 *TTable*、*TQuery* 和 *TstoredProc* 构件进行连接。

在多层体系结构中，一个客户程序至少要有一个 TClientDataSet 构件，它的作用是引入数据集。TClientDataSet 是从 TDataSet 继承下来的，它不需要依赖 BDE。



跟我来

为了创建应用服务器，需要开始一个新项目，并存储它，然后按照以下步骤进行：

（1）在项目中增加一个数据模块“Data Module”。

（2）在数据模块中放置一个或几个连接构件。可以选择 TDCOMConnection、TSocketConnection、TOLEnterpriseConnection、TCorbaConnection、TRemoteServer 或 TMIDASConnection 等构件放到数据模块上。至于究竟选择哪一种 MIDAS 连接构件，这取决于通信协议。

（3）设置有关属性指定和连接应用服务器，这与具体的 MIDAS 连接构件有关。有的 MIDAS 连接构件还有 ObjectBroker 属性，可以指定一个 TSimpleObjectBroker 构件，这个构件可以动态地选择应用服务器。

（4）把一个或几个 TClientDataSet 构件放到数据模块上，并设置 RemoteServer 属性指定一个 MIDAS 连接构件，设置 ProviderName 属性指定应用服务器上的 TDataSetResolver 或 TProvider 构件，这样，客户程序就可以通过 IProvider 接口与应用服务器通信。

（5）把一个 TDataSource 构件放到数据模块上，并设置它的 DataSet 属性来指定 TClientDataSet 构件，再把一个数据控件如 TDBGrid 放到窗体上，设置它的 DataSource 属性以指定 TDataSource 构件。

至此，一个简单的客户程序创建完毕。然而，为了以下目的需要增加代码或者设置属性：

- 管理服务器连接；
- 处理约束；
- 修改或者刷新记录；
- 从应用服务器中获得参数。

1. 连接应用服务器

要建立和维护同应用服务器的连接，客户程序可以使用一个或几个 MIDAS 连接构件，这些构件在构件选项板的“MIDAS”页上可以找到。

使用连接构件可以实现以下功能：

- 确定同应用服务器的通信协议。不同类型的 MIDAS 连接构件使用不同的通信协议；
- 指出如何定位应用服务器所在计算机。识别服务器所在计算机的过程对于不同的协议是不一样。
- 确定应用服务器。

如果不使用 CORBA，则确定应用服务器使用 *ServerName* 或者 *ServerGUID* property 属性。*ServerName* 是你在应用服务器的远程数据模块中所指定的类名。如果服务器已被注册或者已被安装在客户机上，或者连接构件已连上了服务器所在计算机，此时你可以在 Delphi 的属性编辑器的 *ServerName* 属性下拉列表框中选择属性值。*ServerGUID* 指明远程数据模块接口的 GUID 值，可以在类型库编辑器（type library editor.）中选择不同的 GUID 值。

如果使用了 CORBA，应通过 *RepositoryID* 属性指明服务器。*RepositoryID* 属性确定了应用服务器的 factory 接口的 Repository ID，这个 ID 是在调用 *TCorbaVCLComponentFactory.Create*（被自动加入到 CORBA 服务器实现单元的初始化部分）时出现的。

- 管理服务器连接。连接构件常常用于创建和断开连接，以及调用应用服务器接口。

通常，应用服务器与客户应用程序不在同一台计算机中。即使它们在同一台计算机中，你仍需要通过连接构件来指定应用服务器以及所在计算机，并使用应用服务器的接口。

下面就详细介绍这几种 MIDAS 连接构件。

（1）用 DCOM 连接。要使用 DCOM 方式来连接应用服务器，就要用到 TDCOMConnection 构件。TDCOMConnection 构件的 ComputerName 属性用于指定应用服务器所在的计算机。

如果 ComputerName 属性为空，TDCOMConnection 就假设应用服务器与客户程序在同一个计算机上，否则应用服务器在系统注册表中可以找到。反过来说，如果应用服务器没有在客户端注册，而且与客户程序不在同一个计算机上，就须设置 ComputerName 属性。



即使应用服务器在客户端注册了，仍然可以设置 ComputerName 属性来重新指定一个另--台计算机上的应用服务器。ComputerName 属性一般设为计算机的主机名或 IP 地址。

如果指定的主机名或 IP 地址是非法的或者没有找到，TDCOMConnection 就会触发异常。

如果有多个服务器可供选择，最好不要用 ComputerName 属性来切换，而应该加入一个 TSimpleObjectBroker 构件，可以由 ObjectBroker 属性指定 TSimpleObjectBroker 构件。

（2）使用 Socket 连接。客户应用程序可以通过 socket 连接到任何一台具有 TCP/IP 地址的计算机上的应用服务器。这种方式具有应用广泛的优点，因为支持 TCP/IP 环境是相当普遍的，但是，它并不提供安全协议。要使用 Socket 方式来连接应用服务器，就需用到 TSocketConnection 构件。

TSocketConnection 用 Address 属性或 Host 属性来定位应用服务器所在的计算机，前者用于指定 IP 地址，后者用于指定主机名，这两个属性是互斥的，只需要设置其中一个。

此外，还要设置 Port 属性指定端口号，它必须与应用服务器上运行的 Scktsrver.exe 或 Scktsrvc.exe 所使用的端口号一致，缺省值是 211。

提示：可以运行命令行 ScktSrvr.exe（Borland Socket server）来配置 socket 的端口，或者右击任务栏上的 ScktSrvr.exe（如果运行了它的话）来实现。

如果有多个服务器可供选择的话，你应该使用 *ObjectBroker* 属性，而不是 *Address* or *Host* 属性。

（3）使用 OLEnterprise 连接。要使用 OLEnterprise 方式连接应用服务器，就要用到 TOLEnterpriseConnection 构件。当使用 OLEnterprise 时，你可以直接连接应用服务器，也可以使用 Business Object Broker 进行连接。

- 如果直接连接应用服务器，就要设置 ComputerName 属性为应用服务器所在的主机名，就好像 DCOM 方式一样。
- 如果要通过 Business Object Broker 连接应用服务器，就要设置 BrokerName 属性为 Business Object Broker 的名称。

ComputerName 属性和 BrokerName 属性是互斥的，设置了其中一个，另一个就为清空。

（4）使用 CORBA 连接。要使用 CORBA 方式连接应用服务器，就须用到 TCorbaConnection 构件。

对于 CORBA 方式来说，只需要设置 RepositoryID 属性标识 CORBA 数据模块的接口，因为局域网中的智能代理（Smart Agent）会自动定位一个可用的应用服务器。

不过，如果不想让 Smart Agent 自动定位一个应用服务器，而是想指定一个特定的应用服务器，就要设置 HostName 属性为应用服务器的主机名或 IP 地址。

TCorbaConnection 构件可以通过下面两种方式获得应用服务器上 CORBA 数据模块的接口：

- 如果要使用静态联编方式，必须把_TLB.pas 文件（由类型库编辑器生成）加到客户

程序中。静态联编方式不仅能够在编译期进行类型检查，而且运行速度较快。

- 如果要使用动态联编方式，CORBA 数据模块的接口必须用 InterfaceRepository 注册。

（5）代理（Brokering）连接。如果有多个服务器可供选择，最好用 TSimpleObjectBroker 构件来切换，而不要使用 ComputerName 属性。

TSimpleObjectBroker 能够自动维护一个可用的应用服务器的列表。当 MIDAS 连接构件需要连接一个应用服务器时，它就向 TSimpleObjectBroker 提出申请，TSimpleObjectBroker 一般会随机提供一个应用服务器。如果 TSimpleObjectBroker 提供的应用服务器没法工作，TSimpleObjectBroker 会再换一个，一直到 MIDAS 连接构件与应用服务器建立了连接为止。

一旦 MIDAS 连接构件与应用服务器建立了连接，它会自动把应用服务器的有关情况保存到有关属性中，如 ComputerName、Address 或 Host 等，因为 MIDAS 连接构件有可能会断开连接，以后又要再次连接，这时候就不必再向 TSimpleObjectBroker 提出申请。

使用 TSimpleObjectBroker 应指定 *ObjectBroker* 属性。一旦 *ObjectBroker* 属性设定后，*ComputerName*， *Address* 或者 *Host* 值就不会被存储。



在使用 OLEnterprise 或 CORBA 方式的情况下，不要用 TSimpleObjectBroker 构件，因为这两种方式都有自己专门的中介服务。

2. 管理服务器连接

连接构件的主要目的就是定位和连接应用服务器。因为它们管理着连接，所以，你可以使用连接构件来调用应用服务器的接口。

（1）连接应用服务器。为了定位和连接应用服务器，应首先设置连接构件的属性以指定应用服务器。另外，在连接应用服务器之前，任何一个使用连接而构件获得 *IProvider* 接口的客户数据集（TClientDataSet）应该通过设定 *RemoteServer* 属性来指定一个 MIDAS 连接构件，然后再设置 ProviderName 属性指定应用服务器中的 TDataSetResolver 或 TProvider 构件。

当客户程序试图访问 IProvider 接口时，就会自动建立与应用服务器的连接，例如，把 TClientDataSet 的 Active 属性设为 True。不过，在你还未将 TclientDataSet 与连接构件连上以便可以使用 *IProvider* 接口的话，你就该通过 MIDAS 设置连接构件的 Connected 属性来连接或断开连接。

在将要与应用服务器建立连接之前，会触发 BeforeConnect 事件。当建立了与应用服务器的连接后，会触发 AfterConnect 事件。

（2）断开或者修改连接。当进行下列操作时会使连接断开：

- 把 Connected 属性设为 False。
- 关闭客户程序或 MIDAS 连接构件被删除。
- MIDAS 连接构件的 ServerName、ServerGUID、ComputerName、Host 以及 Address 等属性修改后，也会使原有的连接断开，再根据修改后的属性值与新的应用服务器重新建立连接。



尽量不要使用修改 ComputerName、Host 等属性的方式来切换应用服务器，最好用 TSimpleObjectBroker，或者使用几个 MIDAS 连接构件分别建立连接。

与应用服务器的连接将要断开之前会触发 BeforeDisconnect 事件，连接真正断开之后，会触发 AfterDisconnect 事件。可以将相应要处理的代码放入这些事件中。

3. 调用服务器接口

通过 TClientDataSet 的 Provider 属性可以获得 IProvider 接口。其实，大部分情况下并不需要直接调用 IProvider 接口，因为对 IProvider 接口的调用已经封装在 TClientDataSet 的属性和方法中，唯一的例外是，DataRequest 函数只能通过 IProvider 接口调用。当必须使用 provider 接口时，你可以使用 TClientDataSet 的 *Provider* 属性来获得。

当客户应用程序不必调用应用服务器远程数据模块的 *IDataBroker* 接口，你可能已经将你自己的扩展加入到了远程数据模块中了。当你扩展了应用服务器的接口，你就需要有一种方法来调用这些使用连接构件创建连接的扩展。为了能调用扩展，需要使用 *AppServer* 属性。

通过 MIDAS 连接构件的 *AppServer* 属性可以获得应用服务器上远程数据模块的接口，通过此接口可以调用远程数据模块的方法，例如：

```
MyConnection.AppServer.SpecialMethod（x，y）;
```

这种调用方式是动态联编的，也就是说，编译器并不检查 SpecialMethod 的参数，甚至连有没有 SpecialMethod 它都不管。由于编译器不进行类型检查，在运行期实际调用时有可能失败。而且，动态联编没有静态联编的执行速度快。

如果用 DCOM 或 CORBA 作为通信协议连接应用服务器，最好采用静态联编方式来访问远程数据模块的接口，方法就是用特定的接口类型对 AppServer 属性进行强制类型转换。假设远程数据模块的接口叫 IMyAppServer（它的上级是 IDataBroker），程序示例如下：

```
With MyConnection.AppServer as IMyAppServer Do SpecialMethod（x，y）;
```

上面这行代码适合于 DCOM 方式，下面这行代码适合于 CORBA 方式：

```
With IUnknown（MyConnection.AppServer） as IMyAppServer Do SpecialMethod（x，y）;
```

对于 DCOM 方式来说，要使用静态联编方式调用远程数据模块的接口时，它的类型库必须在客户端注册。要注册类型库，可以调用 DELPHI4\BIN 目录中的 TREGSVR.EXE。

对于 CORBA 方式来说，要使用静态联编方式调用远程数据模块的接口时，必须在客户程序中引用类型库编辑器生成的_TLB.pas 文件。

对于 TCP/IP 或 OLEnterprise 方式来说，没法使用真正的静态联编方式来调用远程数据模块的接口，不过，可以通过远程数据模块的调度接口来改善性能，程序示例如下：

```
varTempInterface: IMyAppServerDisp;
Begin
TempInterface := MyConnection.AppServer;
...
TempInterface.SpecialMethod（x，y）;
...
End;
```

要通过调用接口访问远程数据模块时，必须在客户程序中引用类型库编辑器生成的_TLB.pas 文件。

4. 处理约束

SQL Explorer 可以把服务器端的约束和缺省表达式引入到一个数据字典中。应用服务器上的基于 BDE 的数据集可以利用在数据字典中的约束和缺省表达式中。缺省的情况下，服

务器的约束和表达式可以通过应用服务器传递给客户端的数据集，在那里约束和表达式都会影响到用户的数据编辑。当这些约束有效时，而如果用户在客户端输入或修改的数据违反了这些约束的话，则这些数据只在客户端执行，而不会传递到应用服务器端，更不会传递到 RDBMS 端。

由此可见，在客户端纠错可以避免把错误的数据传递给远程数据库服务器，从而节省了网络上不必要的传输，因为这些数据即使传递给远程数据库服务器，也会被退回。

服务器中的约束和表达式的引入是一个比较有价值的特点，这种特点使得开发者在整个平台和应用中保证了数据的完整性。不过，有时候也需要暂时禁止在客户端使用。例如，假设有一项约束需要基于字段的最大值，而客户端一次只能检索若干条记录，也就是说，客户端统计出来的最大值与服务器端统计出来的最大值有可能不一样。即使客户端使用了过滤，也有可能使客户端统计出来的最大值与服务器端统计出来的最大值不一样。上述情况下，就需要暂时禁止使用约束。

要暂时禁止使用这些约束，可以调用 TClientDataSet 的 DisableConstraints，DisableConstraints 的作用实际上是使一个内部的引用计数加 1，当这个引用计数大于 0，就不允许纠错。

要重新使这些约束有效，可以调用 TClientDataSet 的 EnableConstraints，EnableConstraints 的作用实际上是使一个内部的计数减 1，当这个计数减到 0 时，就可以重新允许纠错。

技巧： *如果能成对的出现 DisableConstraints 和 EnableConstraints 的调用的话，这样想什么时候使之有效都较方便。*

5. 修改和刷新记录

当客户程序连接上应用服务器时，客户端数据集就会在内存中建立从应用服务器检索到的数据的副本。如果服务器的约束对客户端的数据集有效的话，用户的数据就会受到约制。用户的修改被客户端的数据集临时存储在一个内部维护的数据更改日志中。用户对数据进行编辑修改后，TclientDataSet 将更改日志作为数据包存储在 *Delta* 属性中，这个数据包中包括更新、删除和插入的记录。为了使修改了的数据永久化，就要向数据库申请更新数据。

（1）修改记录的一般步骤

当客户端使用 *IProvider* 接口向数据库申请更新数据时，应按照以下步骤进行：

① 客户程序调用 ApplyUpdates 函数向应用服务器提出申请，ApplyUpdates 函数将通过 IProvider 接口把 Delta 属性传递给应用服务器。

② 应用服务器收到客户程序的申请后，再向远程数据库服务器提出申请，并且把被远程数据库服务器认为出错的记录暂时保存起来。

③ 应用服务器上的 TDataSetProvider 或 TProvider 构件把出错的记录以 *Result* 数据包返回给客户程序，其中包括错误信息和错误代码。

④ 客户程序收到这些出错的记录后，可以进行核对和修改，然后继续更新。



如果应用服务器端使用 MTS 类型的远程数据模块，就无法提供 IProvider 接口，这种情况下，必须通过远程数据模块的接口直接申请更新数据。

（2）ApplyUpdates 函数

客户端直到使用 *TclientDataSet.ApplyUpdates* 方法让用户修改了数据后，才将数据的副本送至应用服务器。

ApplyUpdates 函数只有一个 MaxErrors 参数，用于指定一个最大错误数，如果出错的记录数超过了这个参数的值，此次更新就停止。如果 MaxErrors 参数设为 0，只要应用服务器发现有一个错误的记录，更新操作就停止。如果 MaxErrors 参数设为–1，当应用服务器发现有错误的记录，就尝试更新下一个记录，等所有的记录都尝试过以后才返回。ApplyUpdates 函数将返回实际遇到的错误数，同时，应用服务器将返回那些有错误的记录。

TclientDataSet 构件负责核对返回的记录。*ApplyUpdates* 方法通过调用 *Reconcile* 将修改送至应用服务器，同时，*Reconcile* 将触发 *OnReconcileError* 事件，如果该事件中有代码的话，则对于每一个出错的记录都触发一次。

（3）核对出错的记录

当应用服务器把出错的记录返回给客户程序时，将在客户端触发 OnReconcileError 事件，这样，就有机会对记录进行核对并修改。所以，你应该在该事件中编写代码，即使你仅仅为了放弃返回的数据记录。

OnReconcileError 事件是这样声明的：

```
TReconcileErrorEvent = Procedure （DataSet: TClientDataSet;
                                   E: EReconcileError;
                                   UpdateKind:TUpdateKind;
                                   var Action: TReconcileAction） of object;
```

其中，DataSet 参数是 TClientDataSet 构件名，通过它可访问数据集中某字段的 NewValue、OldValue、CurValue 等属性，从而分析有关出错的原因。通过 DataSet 参数也能够调用 TClientDataSet 的方法，但不能进行会导致数据被修改的操作，否则，可能会引起死循环。

E 参数是一个指向 EReconcileError 对象的指针，从中可以提取出有关错误信息和导致错误的原因。UpdateKind 参数表示是哪种操作导致了错误，可以是以下值：ukModify（修改）、ukInsert（插入）、ukDelete（删除）。

Action 参数用于决定在退出处理 OnReconcileError 事件的句柄后是否继续更新数据集，可以设为下列值：

- raSkip　表示跳过这条记录，将其仍留在更改日志中；
- raAbort 表示停止整个核对过程；
- raMerge　表示合并修改；
- raCorrect　表示用该事件中所修改的数值替换在更改日志中所做的修改；
- raCancel　表示收回对这条记录的更改，返回到原先所提供的数值；
- raRefresh　表示更改当前的记录以便匹配服务器上的数据。

下面这个程序示例演示了怎样调用 RecError 单元中的一个对话框：

```
Procedure TForm1.ClientDataSetReconcileError（DataSet:TClientDataSet;
        E:EReconcileError;          UpdateKind:TUpdateKind，
        var Action TReconcileAction）;
Begin
    Action := HandleReconcileError（DataSet，  UpdateKind，  E）;
End;
```

（4）刷新记录

客户程序把数据在内存中建立了一个副本，并工作于这个副本，而其他用户有可能已

经修改了数据，也就是说，内存中的数据已不是数据库中的实际数据。

像其他数据集构件一样，可以调用 TClientDataSet 的 Refresh 使内存中的数据是当前最新的。不过，调用 Refresh 前要保证客户端的更改日志中没有记载任何修改，否则会触发异常。

不过，TClientDataSet 的 RefreshRecord 可以不管当前有没有未决的修改，它都可以刷新当前记录，而日志中记载的修改继续保留。



调用 RefreshRecord 有可能带来冲突。因此，调用 RefreshRecord 之前，最好还是检查一下当前是否有未决的修改，如果有的话，就触发异常，程序示例如下：

```
If ClientDataSet1.UpdateStatus <> usUnModified then
    Raise Exception.Create（'You must apply updates before refreshing the current record.'）;
ClientDataSet1.RefreshRecord;
```

6. 从应用服务器上获得参数

客户程序有下列两种情况需要从应用服务器获得参数：

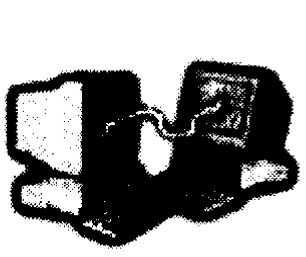
- 客户程序需要知道存储过程的输出参数；
- 客户程序需要初始化 TQuery 或 TStoredProc 的输入参数。

要从应用服务器获得参数，需要调用 TClientDataSet 的 FetchParams 方法，这个函数将返回有关参数并作为 Params 属性的值。

在设计期可以通过用鼠标右键单击 TClientDataSet 构件，然后在弹出的菜单中选择“Fetch Params”命令来获取参数。



无论是在运行期调用 FetchParams 函数，还是在设计期使用“FetchParams”命令，客户端必须已经与应用服务器连接并且获得了 IProvider 接口，因为这些参数是由应用服务器上的 TDataSetProvider 或 TProvider 构件提供的，它们的 DataSet 属性必须指定了 TQuery 构件或 TStoredProc 构件。也可以在客户端设置 Params 属性，然后把参数传递给应用服务器。



定制应用服务器



遇到难题

在进行多层结构的编程时，应用服务器设计的好与坏关系到客户应用程序实现的方便与否。为了能够满足实际的需求，需要能够定制应用服务器。



钥匙在此

MIDAS 的结构是非常灵活的，可以自定义应用服务器以适应实际的需要。如，你可以：

- 扩展远程数据模块的接口；
- 使用自定义的数据集。



跟我来

1. 扩展应用服务器的接口

客户程序与应用服务器之间是通过创建和连接远程数据模块的实例来进行通信的。因此，远程数据模块是应用服务器的核心部件，它提供了应用服务器与客户程序通信的基本接口，除非客户程序直接调用 IProvider 接口。

当在 MTS 中使用事务并及时激活时，通过远程数据模块的接口与应用服务器进行通信显得更为重要了。如果试图绕过 MTS 代理，就会使事务无效，特别是，由于 MTS 类型的远程数据模块可以与状态无关，这时候如果绕过 MTS 代理，有可能导致程序崩溃。

同样，单实例模式的 CORBA 数据模块也是与状态无关的，也存在着上述的问题。

为了使客户程序能够方便地访问 MTS 或 CORBA 数据模块（因为这时候没有 IProvider 接口），必须对远程数据模块的接口进行扩展，添加一些方法让客户程序调用。MTS 或 CORBA 数据模块的接口都是从 IDataBroker 继承下来的。

要向接口中加入新的属性或方法，可以使用以下两种方式：

（1） 在 Delphi IDE 中使用“Edit”菜单上的“Add to Interface”命令。确定是否增加过程、函数、属性或者敲入语法。“单击 OK”按钮，Delphi 4 和 Delphi 5 就会把属性或方法加入到远程数据模块的接口中。

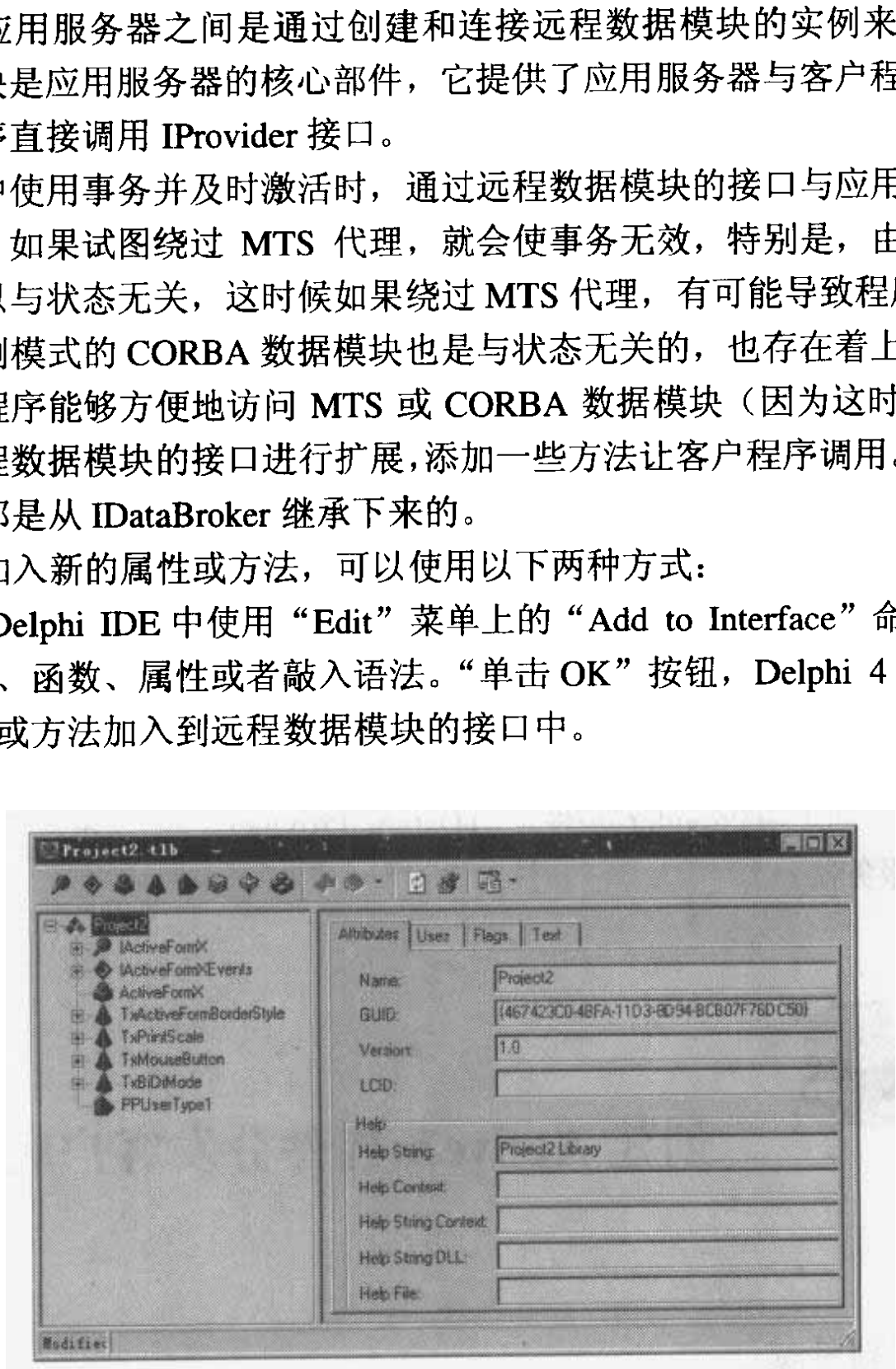


图 20-5 类型库编辑器

（2）使用类型库编辑器。使用“View”菜单上的“Type Library”命令打开类型库编辑

器，如图 20-5 所示。在类型库编辑器中选择接口节点，然后单击工具栏上的“New Method”按钮或“New Property”按钮来加入新成员。加入了一个新的成员后，要设置有关属性。要说明的是，对于 CORBA 类型的远程数据模块的接口来说，许多属性是无效的。

当在接口中添加了新成员时，Delphi 所要做的事情便是决定是否产生基于 COM（*TRemoteDataModule* or *TMTSDataModule*）或者 CORBA（*TCorbaDataModule*）的服务器。

• 对于基于 COM 的远程数据模块（TRemoteDataModule 或 TMTSDataModule）来说，新的成员将出现在接口的实现单元和类型库的描述文件中。

• 对于基于 CORBA 的远程数据模块（TCorbaDataModule）来说，新的成员除了加到接口的实现单元中外，还会自动生成一个_TLB 单元。如果客户程序需要访问基于 CORBA 的远程数据模块，必须引用这个单元。另外，您可以让类型库编辑器生成 IDL 脚本，然后用 Interface Repository 和 Object Activation Daemon 来注册接口。



直到在类型库编辑器中存储或者在 Delphi IDE 中选择“File|Save All”时，TLB 文件才会被存储。

一旦已经将新成员加入到了远程数据模块的接口中，应在所增加的属性或者方法的实现部分加入相应的代码。

客户程序可以通过 MIDAS 连接构件的 AppServer 属性获取远程数据模块的接口，然后通过接口调用远程数据模块的方法。

2. 引入数据

在应用服务器上，一般要使用基于 BDE 的数据集构件来引入数据，由 TDataSetProvider 或 TProvider 构件的 DataSet 属性来指定此数据集构件。不过，缺省情况下，TDataSetProvider 或 TProvider 构件能够用动态生成的 SQL 语句直接与远程的数据库服务器通信，而不需要借助于基于 BDE 的数据集构件，这样做的好处是减少了一个环节的工作。

不过，TDataSetProvider 或 TProvider 构件有时候也需要直接向应用服务器上的数据集构件申请更新数据，因为应用服务器上使用的有可能不是基于 BDE 的数据集，而是 TClientDataSet 或自定义的数据集。这时候需要把 ResolveToDataSet 属性设为 True。

如果你的应用服务器不需要用到 BDE，你应该使用 TDataSetProvider 构件而不是 TProvider 构件。不同于 TProvider 构件，TDataSetProvider 不需要依赖 BDE，从而有利于发布和安装应用服务器。



创建 ActiveX 控件分发客户应用程序



遇到难题

将 Delphi 的分布式数据库结构和 ActiveX 的技术结合在一起，则可以将客户端应用程

序开发成 ActiveX 控件进行分发。



钥匙在此

当你使用 ActiveX 控件分发一个客户端应用程序时，应像为其他多层应用程序创建应用服务器一样创建应用服务器。唯一的限制就是你应当使用 DCOM 或者 sockets 作为通信协议，因为你不能保证客户端机器上运行了 OLEnterprise 或 CORBA 的运行期软件。

在设计客户程序的界面时，要用 ActiveForm 代替一般的窗体。

一旦你已经建立和开发了客户应用程序，在另一台机器上支持 ActiveX 的 Web 浏览器就应该能够访问到这个控件。Web 浏览器要能正确的启动客户程序，需求 Web 服务器正运行在带有客户应用程序的机器上。

当客户应用程序使用 DCOM 协议与应用服务器之间进行通信时，载有 Web 浏览器的机器应能够与 DCOM 工作在一起。如果运行在带有 Web 浏览器的 Windows 95 中时，必须安装 DCOM95，该程序可以在微软网站上下载。



跟我来

1. 为客户应用程序创建 Active Form

（1）因为要将客户应用程序开发成 ActiveX 控件，所以必须有一个与客户应用程序运行在同一个系统中的 Web 服务器。这个服务器可以是已有的 Web 服务器，如微软的 IIS 上或 Netscape 的服务器，或者是自己开发的 Web 服务器。

（2）按照前面所述的步骤创建客户应用程序。

（3）如果你的客户程序中使用了数据模块，在 active form 的初始化部分应该增加创建数据模块的调用。

（4）当你的客户应用程序完成后，编译项目，并选择“Project|Web Deployment Options”，便会出现“Web Deployment Options”对话框，如图 20-6。你要做以下事情：

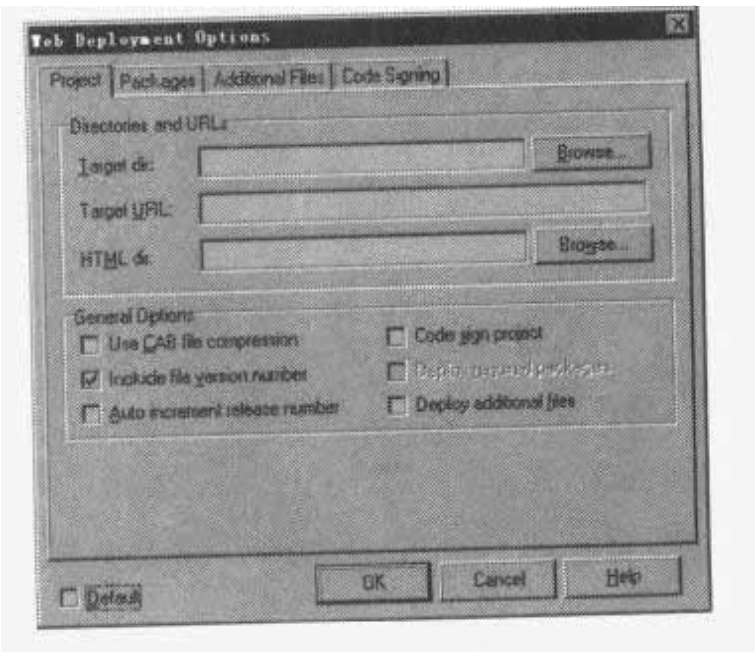


图 20-6　Web Deployment Options 对话框

• 在“project”页指定目标目录、目标目录的 URL 地址以及 HTML 目录。典型的情况，目标目录和 HTML 目录和 Web server 是同一个目录。目标 URL 一般是服务器机器的名字，这个名字是在“网络|DNS”指定的名字。

• 在附加文件页上，选择包含文件 dbclient.dll 和 stdvcl32.dll 并放到客户程序中。

（5）选择“Project|WebDeploy”命令，以 active form 发布客户端应用程序。

任何一个能运行 Active forms 的 web 浏览器，都能运行你的.htm 文件所指定的客户端应用程序，这个.htm 文件是在发布你的客户程序时产生的文件。这个文件有与你的客户程序的项目文件名字相同，并出现在目标目录所指定的目录中。

第二十一章

帮助文件

热点透视

在 Windows 95/98 及 WindowsNT 中，帮助系统中有许多新特性，这些新特性使得帮助界面更为友好，信息材料更为丰富。本章将系统地介绍帮助系统的开发。最后，给定一个应用实例，介绍在 Delphi 应用程序中如何通过各种方式来访问帮助信息。本章的内容主要包括：

- 理解帮助系统的组成
- 进行帮助系统的设计
- 建立帮助系统的主题结构
- 设计帮助项目文件和目录文件
- 调用帮助文件



理解帮助系统的组成



遇到难题

对于一个较为完整的应用程序来说，为用户提供相关的联机帮助是必要的。这些帮助信息可告诉用户应用程序新特点是什么（只适用于升级软件）、功能是什么、如何操作以及有哪些相关主题等等。对于 Delphi 控件的帮助，一般包括这个控件的属性、方法、事件等帮助主题等。作为程序员来说，首先要了解帮助系统是如何构成的。



钥匙在此

在 Windows 操作系统中，帮助系统已经是一个标准的系统，它主要包括以下三个部分：

（1）WinHelp.exe 帮助信息应用程序　这是一个标准的基于 Windows 的帮助信息应用程序，帮助文件信息的显示和解释就是由该程序来实现，即帮助文件浏览器；

（2）帮助文件（.HLP）及目录文件（.CNT）　帮助文件（.HLP）是我们要开发的经编译生成的帮助文件，目录文件中存放的是帮助信息的主题目录层次结构；

（3）WinHelp 函数　这是应用程序中调用的 WinHelp 函数，通过它来调用帮助信息。



跟我来

1. 帮助文件

在 Windows 95/98 以及 WindowsNT 环境中开发帮助系统，最终的产品是帮助文件和帮助目录文件，即 filename.hlp 和 filename.cnt。

（1）帮助文件 帮助文件的文件扩展名是.hlp，它包含了与具体应用程序相关的所有帮助信息。它是由帮助项目文件以及主题文件经过编译和连接而成的。

（2）帮助目录文件 帮助目录文件的文件扩展名是.cnt，它包含了帮助信息的主题目录层次结构,为使用者通过帮助主题来查找相应的主题信息提供了入口。

帮助文件生成后，一般是与应用程序结合在一起使用，但也可以直接打开而单独使用。

例如，在 Delphi 的开发环境中，选择设计界面中的某个控件然后按“F1”键，相应控件的帮助信息框就弹出了。这是 Delphi 开发工具与 Delphi 的帮助文件结合在一起使用的例子。

又如，也可以在 Windows 的【资源管理器】窗口中，直接双击某个帮助文件的图标，Winhelp 应用程序就会按缺省的方式打开该帮助文件。

2. 帮助信息的构成

（1）帮助主题对话框

当用户直接双击某个帮助文件时，首先显示的往往是一个帮助主题对话框。

如果该帮助系统中包含了帮助目录文件时，帮助主题对话框中则有一个【目录】选项卡。除此之外通常还会带有一个【索引】选项卡，有时根据需要，还会加上【查找】选项卡或其他选项卡。

【目录】选项卡中的主要部分是一个树形视图列表框，它把各条帮助主题标题按照层次结构有机地组织在一起，使得用户在查阅帮助信息时，就如同在翻阅一本书，可以逐级章节结构追溯下去，找出所需要的主题来，如图 21-1 所示。

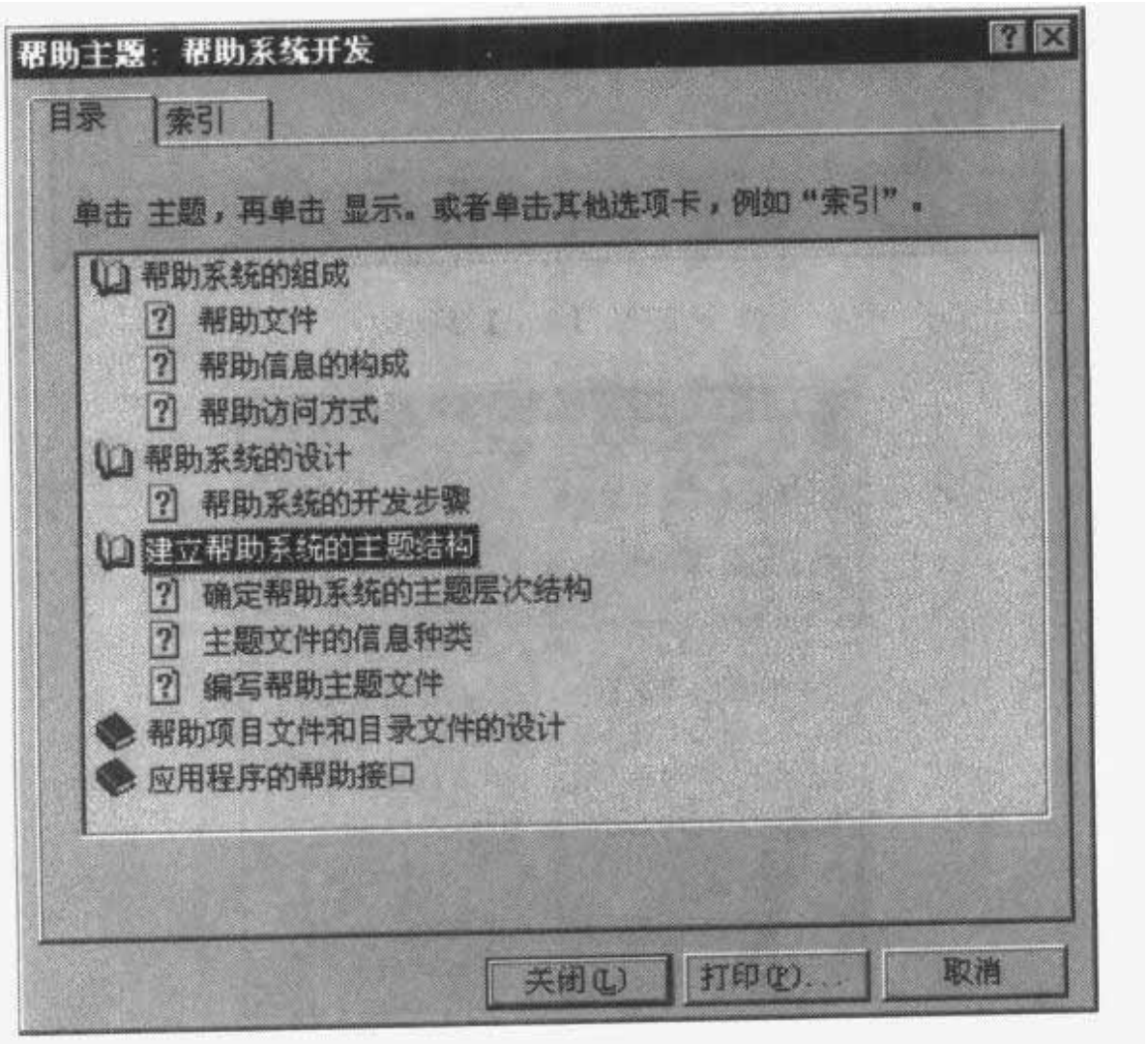


图 21-1 【目录】选项卡

【索引】选项卡中列出了与帮助主题链接在一起的关键词索引列表。用户可以在编辑控件中输入关键词或从列表框中选择关键词来找出相应的帮助主题，如图 21-2 所示。

（2）帮助主窗口

帮助主题主窗口是显示帮助主题信息的主要场所，它是一个标准的帮助窗口，通常包含标题栏、菜单栏、按钮工具栏、滚动条以及帮助信息内容显示区等几个部分，其中信息内容显示区又往往由不可滚动区与可滚动区组成，不可滚动区主要显示主题的标题，而滚动区则显示帮助主题的正文，一个帮助文件只显示一个帮助主题主窗口，如图 21-3 所示。

（3）帮助信息辅助窗口

帮助信息辅助窗口是由用户自定义的非标准窗口。它通常不带有完备的菜单和工具栏。而只是根据具体需要添加某些必要的命令供用户使用。帮助信息辅助窗口具有灵活实用的特

点，它往往作为一个临时性的窗口使用，如图 21-4 所示。

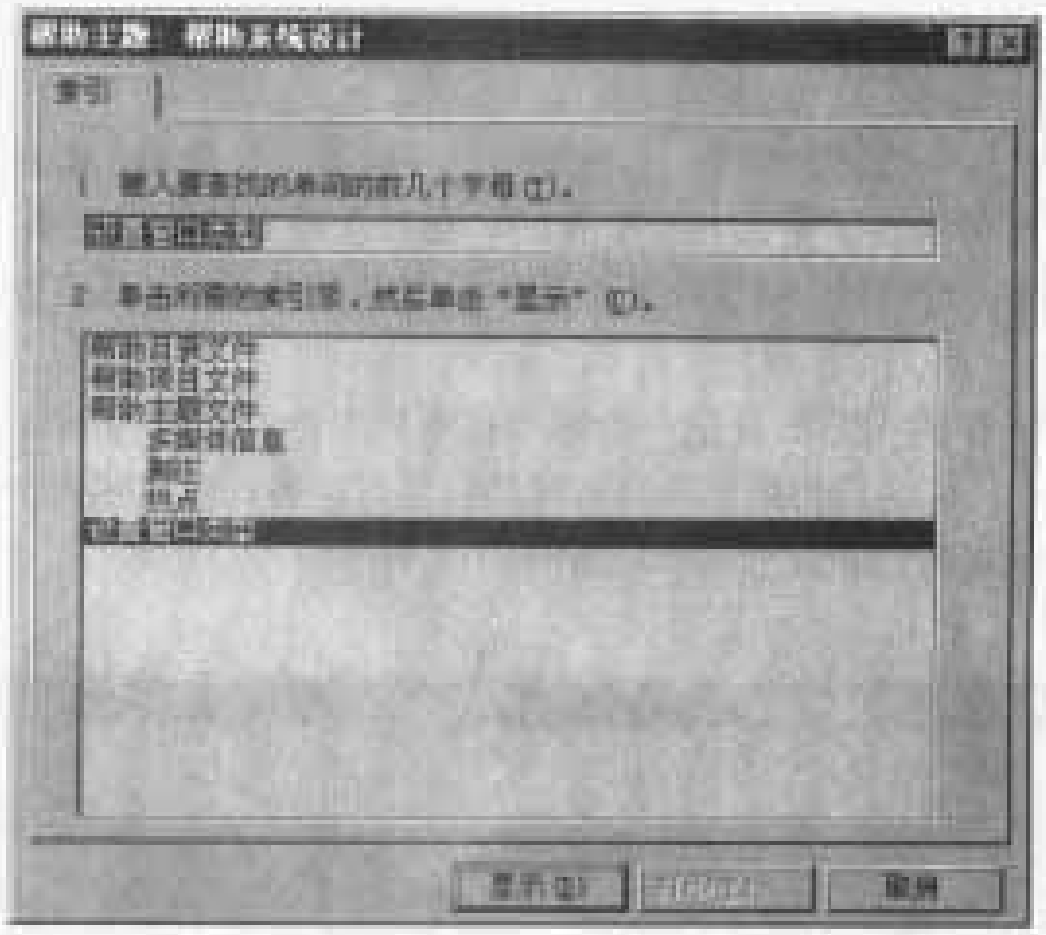


图 21-2 【索引】选项卡



图 21-3 帮助主题主窗口

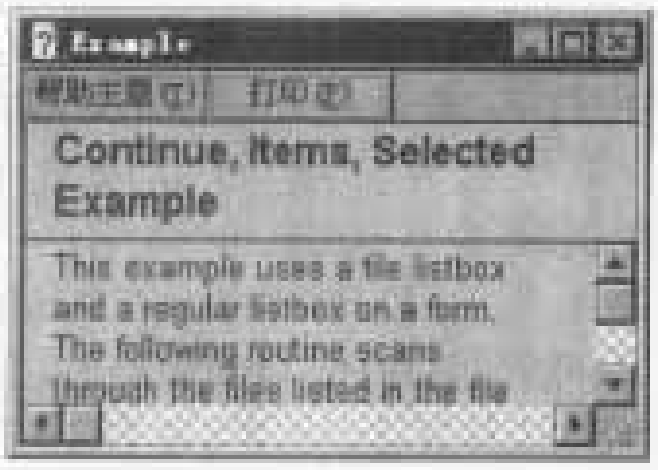


图 21-4 帮助信息辅助窗口

一个帮助文件可同时打开多个辅助窗口（最多可达 9 个）。

（4）弹出式帮助窗口

弹出式帮助窗口的外观最为简练，它不同于一般窗口，不带标题栏和菜单栏等，而且往往只显示一小段简短的文字。它是一个临时性的窗口，不能长时间保留在屏幕上，一旦按下任意键或单击鼠标，它就会自行消失了。这种类型的帮助窗口一般用于名词解释或一些简短的注释。

3. 帮助访问方式

在使用应用程序时，要想访问帮助文件，一般来说有四种方式：访问主题目录、搜索关键词、直接定位主题以及弹出式窗口帮助。其中后两种都是与上下文相关（context-sensitive）的。

（1）访问主题目录

这种访问方式是先打开帮助主题对话框（通常是通过菜单命令实现的），然后在该对话框的【目录】选项卡中逐级找出所需的帮助主题，最后单击【显示】按钮，或者双击该主题项，此时就会显示相应的帮助主题主窗口。

（2）搜索关键词

打开帮助主题对话框，切换到【索引】选项卡，用户直接敲入关键词或者在关键词列表框中选择一个关键词，然后就会显示关键词所对应的帮助主题信息窗口。

（3）直接显示帮助主题主窗口

在应用程序中，通过单击【帮助】按钮或按 FI 键等方式打开帮助系统时，帮助系统就会结合当前应用程序的上下文环境，直接定位到相关的帮助主题，并显示出其帮助主题主窗口。

（4）显示弹出式窗口

在应用程序的窗口（尤其是对话框），单击标题栏上的帮助按钮，然后单击其控件对象，就会弹出一个弹出式窗口，给出简洁的帮助信息。或者用鼠标右键单击某对象，然后在弹出来的菜单中选择“这是什么？”，也会弹出弹出式窗口来。当鼠标在任意位置单击时，该弹出式窗口将会自动消失。



进行帮助系统的设计



遇到难题

对于程序员而言，最关心的是如何开发一个自己的帮助系统呢。



钥匙在此

为了开发一个好的帮助系统，一般得完成以下一些步骤：

（1）规划帮助系统的内容。

收集资料，获取设计帮助系统所需的各种文档，如软件产品的需求规格说明、操作手册以及用户手册等，并且熟悉软件产品。根据用户对象的类型（用户对象可分为计算机初学者、应用程序初学者、有一定经验的用户以及专业用户等几个级别），规划帮助系统的内容。当确定了用户对象的类型后，有的放矢地规划相应的内容；

（2）建立帮助系统的主题结构，并编写主题文件以及相关的图形文件等；

（3）创建和编辑项目文件、主题目录文件；

（4）在应用程序中添加代码，访问帮助文件。



跟我来

如果要设计一个帮助系统，请按照上述提到的步骤进行设计。



建立帮助系统的主题结构



遇到难题

这是进行帮助系统设计的第二步。也就是说如何建立一个帮助系统的大纲。



钥匙在此

1. 确定帮助系统的主题层次结构

在收集了必要的资料，规划完帮助系统的内容后，应确定有哪些帮助主题，并将这些帮助主题组织成层次结构（即目录结构）。从主题目录结构中查找主题是一个从一般到特殊的过程。用户通过打开目录，一级一级地往下找，直到找到所需的主题。

在确定主题的层次结构时应该注意以下几点：

（1）目录层次不宜太深，同一目录下的子项不宜太多。这样便于用户较快地找到主题。

（2）同一层次的相关主题最好能排好顺序，以适合于用户的线性测览，使得用户可避开目录结构，直接向前或向后连贯地阅读相关主题。

（3）主题中可包含着跳转特性，但次数不宜过多。用户不一定非得逐级查找主题，也不见得总是连贯地阅档主题，用户的阅读方式往往是跳跃和联想的。因此，主题中还应该包

含着跳转特性，使用户直接从一个主题跳转到另外几个相关节题上去，这种方式适合于人的阅读习惯，同时也提高了阅读效率。但是如果跳转的次数太多了，那又会令人如走迷宫，理不清头绪，不便于查找主题。

总之，以上无论采用哪一种结构，都要考虑到效率，尽可能缩短目录的搜索路径，尽可能减少相关主题跳转次数。

2. 主题文件的信息种类

帮助主题中包括的信息材料种类是丰富多彩的，可以是文字、图形，可以是多媒体信息，也可以是可跳转的热点。下面简要地对这些信息种类作些介绍。

. 文字

文字是帮助信息的主要载体，它又包括了主题的标题文字和正文。标题文字一般位于主题起始处的不可滚动区域，而正文落在下面的可滚动区域中，帮助主题中的文字可以有不同的字体、尺寸以及颜色等。

. 图形及多媒体信息

帮助主题中往往加有图形，它能弥补文字的不足，为用户提供直观而丰富的信息，同时在帮助主题中还可嵌入多媒体信息，如发出一段声音、显示动画等等。但多媒体信息必须保存在帮助文件之外的文件中，并通过外部命令来访问。

. 脚注

在帮助文件中，用户是看不到脚注的，但对帮助文件的开发者来说，脚注是非常重要的一个环节。每个主题都有或多或少的几条脚注，通过使用脚注可以定义帮助主题的标识符（上下文字符串）、定义主题的关键词以及浏览顺序等等。

. 热点

热点是帮助主题中的一些特殊点，它可以是文字，也可以是图形，还可以是图形的一部分。当用户单击它时，往往发生跳转，或者弹出一个弹出式帮助窗口，或者执行一个外部命令。作为热点的文字往往加有了下划线，或者颜色与众不同。当鼠标移动到热点上时，光标就会改变形状，一般变为手形光标。

3. 编写帮助主题文件

帮助主题是一种多文本格式（RTF）文件，有很多较好的工具能生成 RTF 文件，如 Help&Manul、HelpMagic 等软件都是一些较好的软件，是所见即所得（WYSIWYG）的设计工具，但为能很好的了解帮助指令和 RTF 的具体格式，我们还是采用了典型的字处理软件 Microsoft Word 来产生 RTF 文件。

（1）帮助主题

每个帮助主题都是一个信息单元，它基本上包括了三个方面的内容：信息材料（包括标题、正文、图形和多媒体信息等）、脚注以及热点。具体说来，帮助主题除了要有文字外，还必须有一个唯一的标识符（也称为主题上下文），还可以包含图形、多媒体信息以及各种附加的关键词、热点及其他标志。

（2）帮助主题文件

帮助主题文件必须存为 RTF 格式的文件，虽然用普通的文字编辑程序就可以编写出主题文件，但十分不方便，尤其是对于中文帮助主题文件而言更是如此。编写 RTF 文件的工具较多，如， HelpMagic、Emeditor 等等， 较为典型的是 Microsoft Word，它可以较为方便快捷的实现帮助主题文件的编写。

一个帮助主题文件至少得包含一个帮助主题，通常总是包含多个主题，每个主题之间必须以分页符分开。

（3）帮助语句

帮助命令主要分为帮助语句和帮助宏两种。帮助语句用来描述帮助文件中文字和图形的格式和位置等，是多文本（RTF）命令的扩充子集，而帮助宏则定义当帮助文件被打开和显示时采取的一系列动作，如执行菜单命令、创建自定义按钮以及链接主题等。帮助语句可分为两类：RTF 命令和帮助主题命令。

对于 RTF 命令，每条语句都是以反斜线字符（\）开始，后接语句名称，再后面可添加语句参数，它们用空格符分隔开来，许多语句都只作用于特定范围，这个范围就得用花括号（{}）括起来，它把语句及其起作用的文字或图形括在一起，形成一个相对独立的组。例如，\footnote 语句表示添加一条脚注，而“#{\footnote topic1}”则是添加一条标记为#的脚注，其中脚注文字为 topicl。这条脚注是用来定义主题的标识符（上下文字符串）的。现在的开发者很少直接使用这种 RTF 语句，因为在 Microsoft Word 等字处理软件中，可以通过编辑直观地来实现 RTF 命令，就如在可视化开发环境中设置控件的属性一样，而且所见即所得。例如，对于上面的脚注语句，在 Microsoft Word 中，间接通过“插入”|“脚注和尾注”菜单命令，就可以添加这条＃脚注。

帮助主题命令则不能通过字处理编辑来等价实现（除了使用其他开发工具），必须直接使用。各命令语句不用以反斜线字符（\）开始，但也得用花括号（{}）括起来。例如，bmc 语句用来嵌入一幅图形，为了嵌入一幅位图在主题文件中，请在 RTF 文档键入下列语句：

```
{bmc  bmpfilename.bmp}
```

其中 bmpfilename.bmp 为位图文件的文件名。

表 21-1 列出部分常用的帮助主题命令，其他主题命令可以参考“Help Workshop”软件的帮助文件。

表 21-1　　帮助主题命令

语 句	描　　述
Bmc	在文字中显示位图（居中对开）
Bml	左对齐显示位图
Bmr	右对齐显示位图
Button	在帮助主题文件上插入一个按钮
Mci	在帮助主题文件中播放多媒体文件
Emc	允许个别动态链接库（DLL）在文字中间绘制一个嵌入帮助主题的窗口
Eml	允许 DLL 以左对齐方式绘制窗口
Emr	允许 DLL 以右对齐方式绘制窗口



跟我来

由以上内容可知，每个帮助主题主要包含了三个元素：信息材料、脚注以及热点。编写帮助主题文件的步骤主要如下：

（1）在一个能产生 RTF（Rich text format）文件的字处理器或文本编辑器中新建一个文件；

（2）编写单独主题，每个主题之间以分页符隔开；

（3）在文件的相应位置插入适当的脚注；

（4）以 RTF 格式存储文件；

（5）以热点或者图形联接主题；

（6）通过帮助主题命令或直接方式插入图形或多媒体信息至主题中。

注：在 Word 中插入脚注的步骤是：

（1）单击注释参考标记的插入点；

（2）单击“插入”菜单中的“脚注和尾注”命令；

（3）单击“脚注”选项按钮。在编号方式中选择自定义标记，在编辑框中输入适当的脚注符号。如，主题标题的脚注符号为“$”，主题标识符为“#”等等；

（4）在脚注窗格中输入注释，然后单击文档中任意位置以便继续处理正文。

1. 脚注

脚注是一条帮助语句，它用来指定一些与特定主题相关的附加信息，如主题标识符、主题索引关键词、浏览序号以及连编标记等等。这些信息并不会直接显示在帮助文件窗口中，最终用户是看不见它们的，但是对于帮助文件的开发者来说，脚注具有非同寻常的作用，它是 WinHelp 控制和管理帮助文件的关键所在。

每个主题一般都有一条主题标识符脚注。另外，根据需要，还可在帮助主题中加入不同类型的脚注，表 21-2 列出了几种常用的脚注类型。

表 21-2　　脚注类型

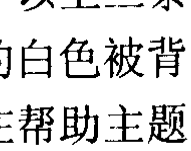
脚注类型	脚注标记	含义
主题标识符	#	定义主题标识符，除了基本主题标识符（定义在主题起始处）之外，还可以在其他地方定义附加的主题标识符
关键词	K	指定主题在索引和 KLink 宏中使用的关键词
A-关键词	A	指定用于标识 ALink 宏中使用的主题关键词
主题标题	$	给出主题的标题，该标题出现在主题【查找对话框中】、【书签对话框】、【历史窗口】或者【查找选项卡】中
浏览顺序号	+	决定用户浏览主题的顺序
入口宏	!	指定主题入口宏，当用户显示该主题时，将自动运行该宏
窗口类型	>	指定显示本主题时所采用的窗口类型
连编标志	*	指定主题的连编标志，以便项目文件确定最终是包括还是排除本主题，如果使用了连编标志，那么这条脚注必须放在主题最首位置
注释	@	加写开发者的注释，仅供内部参考，不显示给用户

指定主题关键词

主题关键词由 K 脚注给出，一个主题可以有多个关键词，每个关键词之间须用分号（;）分隔开来，每个关键词的长度不能超过 255 个字节。

用户还可以使用主级和次级关键词进行索引，例如在“帮助主题文件”这条索引关键词之下，还有三条次级关键词“脚注”、“热点”和“多媒体信息”。这要求在主题文件中编写 K 脚注时，先写上主级关键词，打上分号结束，紧接着后面再写一次主级关键词，加上逗号（,）和空格，再写次级关键词，最后加上分号，如下所示：

帮助主题文件; 帮助主题文件, 脚注; 帮助主题文件, 热点; 帮助主题文件, 多媒体信息;



以上各逗号（,）之后必须有一个空格字符才行。

2. 插入图形

帮助文件支持的图形格式有位图文件 BMP 和 DIB、元文件 WMF、多热点图形 SHG 以及多分辨率图形 MRB 等格式。

在帮助主题文件中插入图形的方法有两种，一种是直接把图形插在主题文件中，另一种是给出图形文件的名称，间接地把图形嵌入到主题文件中。对于位图，这两种方法都是可行的，但对于 SHG 或 MRB 图形，则只能使用第二种办法。

要间接插入图形，可用的帮助主题命令有表 21-3 所示的几条。

表 21-3　　插入图形文件的命令

帮助主题命令	含义
Bmc	图形居中对齐，文字可在图形两侧环绕
Bml	图形向左对齐，文字可在图形右侧环绕
Bmr	图形向右对齐，文字可在图形左侧环绕

以上三条命令都可以后接参数 t，该参数的作用是使得图形在帮助主题中显示时，图形中的白色被背景色所取代。不过，这个透明特性只适用于 16 色图形。例如，下面的命令将会在帮助主题中加上一幅名为 Building.wmf 图形：

{bmct Building.wmf}

有时，考虑到用户在查看帮助主题时，显示模式往往不一致，为了保证好的显示效果，可以预先给出几幅内容相同但颜色数目不同的图形。在显示时，帮助文件会根据当时采用的显示模式，显示其中最合适的一幅。在使用上述图形命令时，把相关的多幅图形放在一个命令语句中，彼此用分号隔开，就可以实现这个要求。例如：

{bmr Flower.wmf;Building}

3. 设置热点

对于多热点图形文件，在引入到帮助主题之前就已经设置好了热点，不过除此以外的文字和图形上都可设置热点。

热点的功能有三种：一是实现主题跳转（即交叉链接），当用户单击热点时，帮助窗口就显示另一个主题，这种热点称为跳转式热点；二是链接到一个弹出式主题上，当用户单击热点时，则显示一个弹出式的帮助窗口，而原有的主题并不消失，这种热点称为弹出式热点；三是执行一条帮助宏，这种热点称为宏热点。

（1）跳转式热点

创建跳转式热点的步骤如下：

① 把插入符定位在要作为热点的文字或位图之后，然后输入目标主题的标识符；

② 选择作为热点的文字或图形，把它加上双下划线；

③ 选择目标主题标识符，使之具有隐藏的字符风格。

例如，下面就是一个图形跳转式热点，当用户单击该位图时，将跳转到主题标识符为“帮助系统的组成”的帮助主题上：

{bmc structure.bmp}帮助系统的组成



为了便于说明，这里约定本书中在文字下用加波浪线的方法表示该文字具有隐藏风格的文字。



在热点与目标主题标识符之间不能留有空格。

对于文字热点，当用户打开帮助文件时，热点文字呈现绿色，并且加有下划线。

技巧：如果不想改用绿色显示热点文字，而想保留其原有颜色，那么应在目标主题标识符之前加上一个星号（*）；要是在目标主题标识符之前插入了百分号（%），那么最终显示出来的热点文字既不会改成绿色，又不会加下划线。例如：

{bmc structure.bmp}*帮助系统的组成

{bmc structure.bmp}%帮助系统的组成

技巧：如果想要指定目标主题显示时采用的窗口，那么就应该在目标主题标识符之后加上一个大于号（>），并加上窗口类型的名称。例如：

{bmc structure.bmp}帮助系统的组成>帮助文件

（2）弹出式热点

设置弹出式热点与设置跳转式热点的步骤相同，需要改动的只是把作为热点的文字或图形的字符风格设为下划线，而且是双下划线。

（3）宏热点

设置宏热点与设置跳转式热点的方法也类似，不同的只是将那里的目标主题标识符改为感叹号（!）加宏命令。例如：

{bmct refer.bmp}!klink （窗口外观属性,0, “”,实践窗口）

由于 Klink 和 Alink 这两条宏在宏热点中经常用到，它们的用法比较相似，这里只简要地给出 Klink 宏的用法，Klink 宏的语法说明如下：

Klink（keyword[;keyword][,type[,topic_id[,window_name]]]）

提示：上面的方括号表示其中的参数可用可不用，以下同。

其中 keyword 参数是必须带的参数，它用来指定要链接的关键词。如果有多个关键词，则必须用分号（;）隔开。type 参数用来指定在找出一个或多个关键词之后的后续动作。通常采用缺省方式（0），即显示一个【已找到的主题】对话框，列出各主题的标题。

提示：如果为某个主题定义了关键词，那么最好先定义该主题的标题。

要是一个带有关键词的主题也没有找到，那么就显示由 topic_ID 参数所指定的主题标识符的主题。最后一个参数 window_name 用来指定显示主题所采用的窗口类型。

4. 按钮热点

button 帮助主题命令用来在帮助主题中插入一个与帮助宏相关联的按钮，当用户按下按钮时，将运行指定的宏。这条命令所带的参数为标签文字和宏，它们之间用逗号（,）隔开。如果有多对标签文字和宏，那么用分号（;）分隔。例如，下面的命令将显示一个按钮：

{Button 相关主题, ALink（帮助信息的构成， 0，“”， main）}

标签文字参数可以不要，但逗号不要省略。例如，把上面的按钮换一种方式来显示：

{Button，ALink（帮助信息的构成，0。“”，main）}相关主题%!ALink（详细操作，0,“”,main）

当有多个按钮排列在一起，而标签文字参差不齐时，采用上面这种方式更合适些。

5. 多媒体信息

与一般的文字和图形相比，多媒体信息更显得生动形象。要是在帮助主题文件中适当地加入多媒体信息，将使得帮助信息更富于启发性，可以为帮助文件增色不少。

在帮助主题文件中使用的多媒体信息主要是声音（WAV 文件）和动画（AVI 文件），它们可以嵌入到任何主题中（包括弹出式主题）。帮助主题文件通过使用 mci 命令来加入多媒体文件。如表 21-4 所示是使用 mci 命令时可选用的参数选项。

表 21-4　　mci 命令的参数及其含义

可选项	含　义
NOPLAYBAR	不显示播放工具条
MOMRNU	如果有播放工具条，则不显示菜单按钮
PLAY	当主题显示时，自动开始播放
REPEAT	循环播放
EXTERNAL	多媒体文件存放在帮助文件之外

例如，添加下列帮助主题命令，在用户显示该主题时，将自动开始循环播放外部动画文件 jxg.avi，且回放区不带播放工具栏：

{mci EXTERNAL PLAY REPEAT NOPLAYBAR, jxg.avi}



设计帮助项目文件和目录文件



遇到难题

这是进行帮助系统设计的第三步。



钥匙在此

帮助项目文件（.hpj）由 Help Workshop 创建，并经 Help Workshop 编译成帮助文件（.hlp），这如同在 Delphi 中开发应用程序的项目文件一样，编译程序把帮助主题文件以及相关的图形文件结合起来转变为帮助文件的全部信息。

帮助目录文件（.cnt）也由 Help Workshop 产生，它保存了在帮助主题对话框的【目录】选项卡下显示目录的层次信息。



跟我来

这里以创建 HelpGuide 帮助文件为例，介绍帮助项目文件和目录文件的创建和设置，并最后编译和连接为帮助文件。

1. 新建帮助项目文件和目录文件

创建帮助项目文件的步骤如下：

（1）打开 Help Workshop，选择“File”菜单的“New”命令；

（2）在弹出的对话框中双击“Help Popect”；

（3）输入项目文件的名称，本例为 HelpGuid。

创建帮助目录文件的步骤与上类似：

（1）打开 Help Workshop，使用“File”菜单下的“New”命令；

（2）在弹出的对话框中双击“Help Contents”；

（3）输入目录文件的名称，本例为 HelpGuid。

注：

Help Workshop 是一个用于产生帮助文件（.hlp）、编辑项目文件和帮助目录文件的应用程序，生成并测试帮助文件，集编辑、编译、连接以及测试于一体。Help Workshop 将项目文件（.hpj）中的信息与主题文件（.rtf）、位图和其他的资源组合成 Microsoft Windows 的帮助程序（WinHelp.exe）能浏览的帮助文件。Help Workshop 的执行程序为 hcw.exe，这个程序位于“\Borland\Delphi4\Help\Tools”的目录下。

2. 编辑帮助目录文件

当创建目录文件之后， Help Workshop 窗口如图 21-5 所示，窗口中的主要部分是一个列表框，它用来显示要编辑的帮助目录，呈树状结构。这个目录最终将出现在帮助主题对话框的【目录】选项卡下。

窗口右边是几个命令按钮，“Add Above”和“Add Below”都是用来往目录结构中添加新的目录题头或主题目录项，只是前者将加在当前所在目录项之前，而后者将加在当前所在目录项之后。如果要调整某目录项在目录层次结构中所处的位置，那么就该使用“Move Right”或“Move Left”按钮。“Remove”按钮用来删除某个目录项。而“Edit”按钮则用来编辑现有的目录项。

在添加或编辑目录项时，就会弹出如图 21-6 所示的对话框。若是添加目录项，首先要在对话框上部的四个单选按钮中选择一项，确定该目录项是什么。“Heading”表示该目录项是一个目录章节题头；“Topic”表示该目录项是主题目录项，以后程序运行时如果用户双击主题目录项，就会显示相应的帮助主题；“Macro”表示该目录项是一条宏；而选择“Include”则可以把另一个目录包含到本目录文件中来。

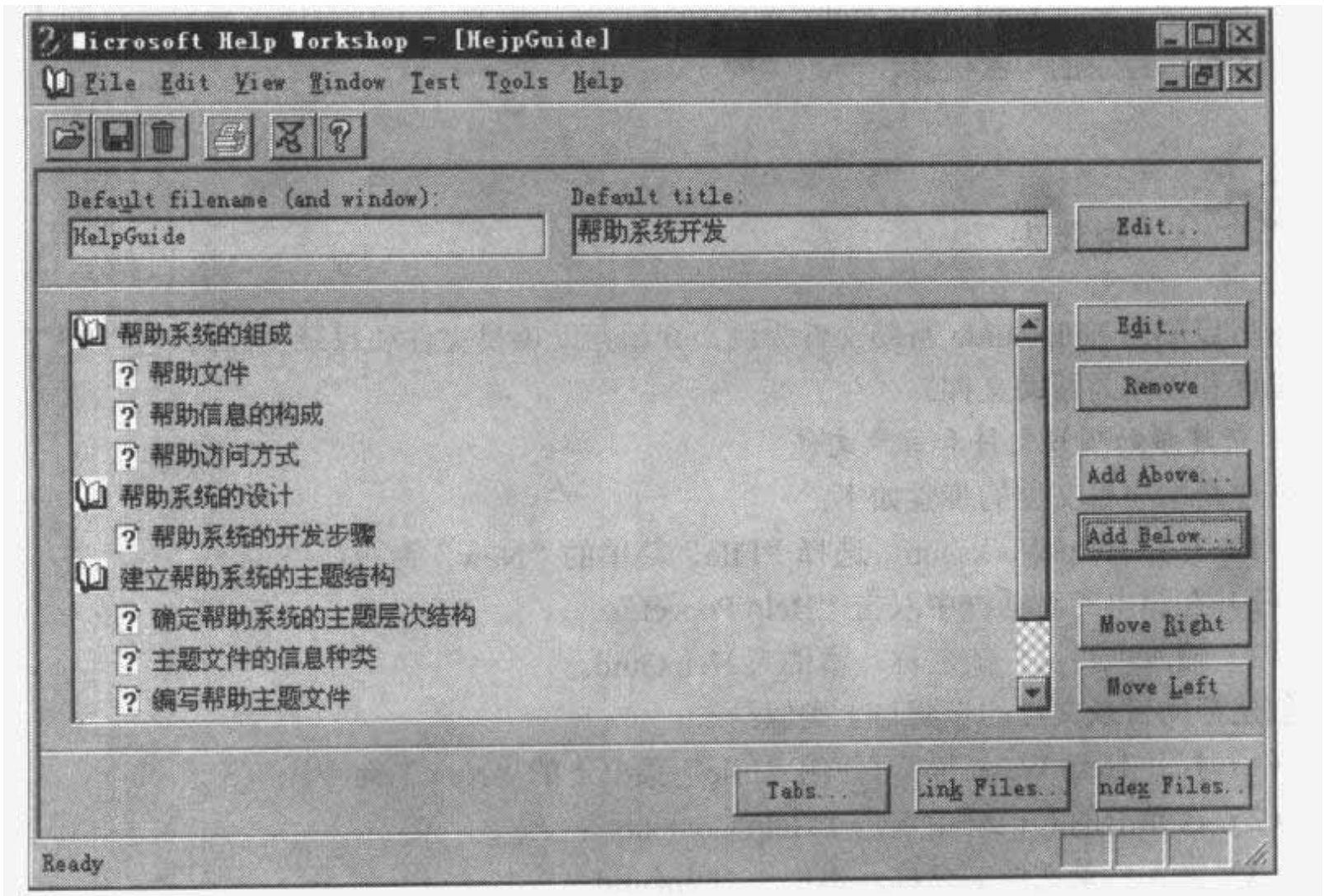


图 21-5　编辑帮助目录文件

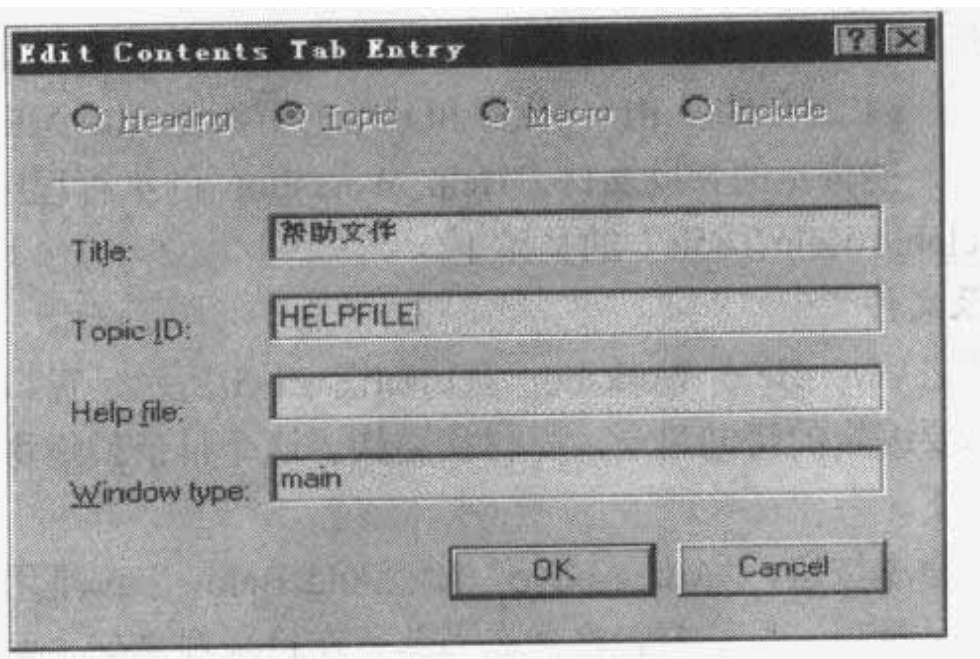


图 21-6　编辑目录项

下方的“Title”编辑框用来输入目录项文字。如果是主题目录项，那么应该在“Topic ID”编辑框中输入对应的主题标识符。在“Topic ID”编辑框中输入的也可以是一条或几条宏，以后程序运行时如果用户单击该目录项，将执行这些宏。接着下面的两个编辑框分别用来指定主题所在的帮助文件以及显示主题时采用的窗口类型。

3. 设置基本的帮助项目选项

在创建或打开帮助项目文件后，Help Workshop 就会打开如图 21-7 所示的窗口。工作区中显示帮助项目的内容，用户不能在此直接修改帮助项目文件的内容，但如果要修改某一项，只须双击该行，然后在打开的对话框中进行间接编辑修改。

帮助项目文件的内容分为许多节，各节的节名由方括号括住，接下来的几行是该节中

包含的选项设置。帮助项目文件中可加入注释语句，它是由分号（;）开始的语句。

帮助项目文件中各节都对应了一定的用途。表 21-5 列出帮助项目文件中可能出现的节。

表 21-5　　项目文件中的节

节	作　用
【FILES】	指定连编时要包括进去的主题文件，本节是不可少的
【OPTIONS】	指定控制连编处理过程的各选项，本节通常不可少
【MAP】	把主题标识符与上下文号码关联起来，以便给程序调用
【WINDOWS】	定义帮助文件中用到的帮助主窗口及辅助窗口的特性
【CONFIG】	指定作者定义的帮助宏，并注册 DLL 以及作用宏调用的 DLL 函数
【CONFIG：name】	作用同上。但仅适用于单个窗口的配置
【BAGGAGE】	列出要嵌入到帮助文件中的文件
【MACROS】	指定当用户从索引表中选择指定关键词时要运行的宏
【INCLUDE】	指定将要嵌入到帮助文件中的主题的连编标记
【EXCLUDE】	指定将要排除在帮助文件之外的主题的连编标记
【ALIAS】	给一个主题标识符赋一个或多个其他主题标识符

窗口右边有一列按钮。右下角的“Save and Compile”按钮的作用是在完成编辑后，保存并编译项目文件。其他按钮都是用来添加或修改项目文件中的选项的。每个按钮基本上对应于项目文件中的一节。如“Options”按钮对应于 OPTIONS 节，“Files”按钮对应于 FILES 节，“Wndows”按钮对应于 WINDOWS 节，“Config”按钮对应于 CONFIG 节，“Data Files”按钮对应于 BAGGAGE 节，如此等等。

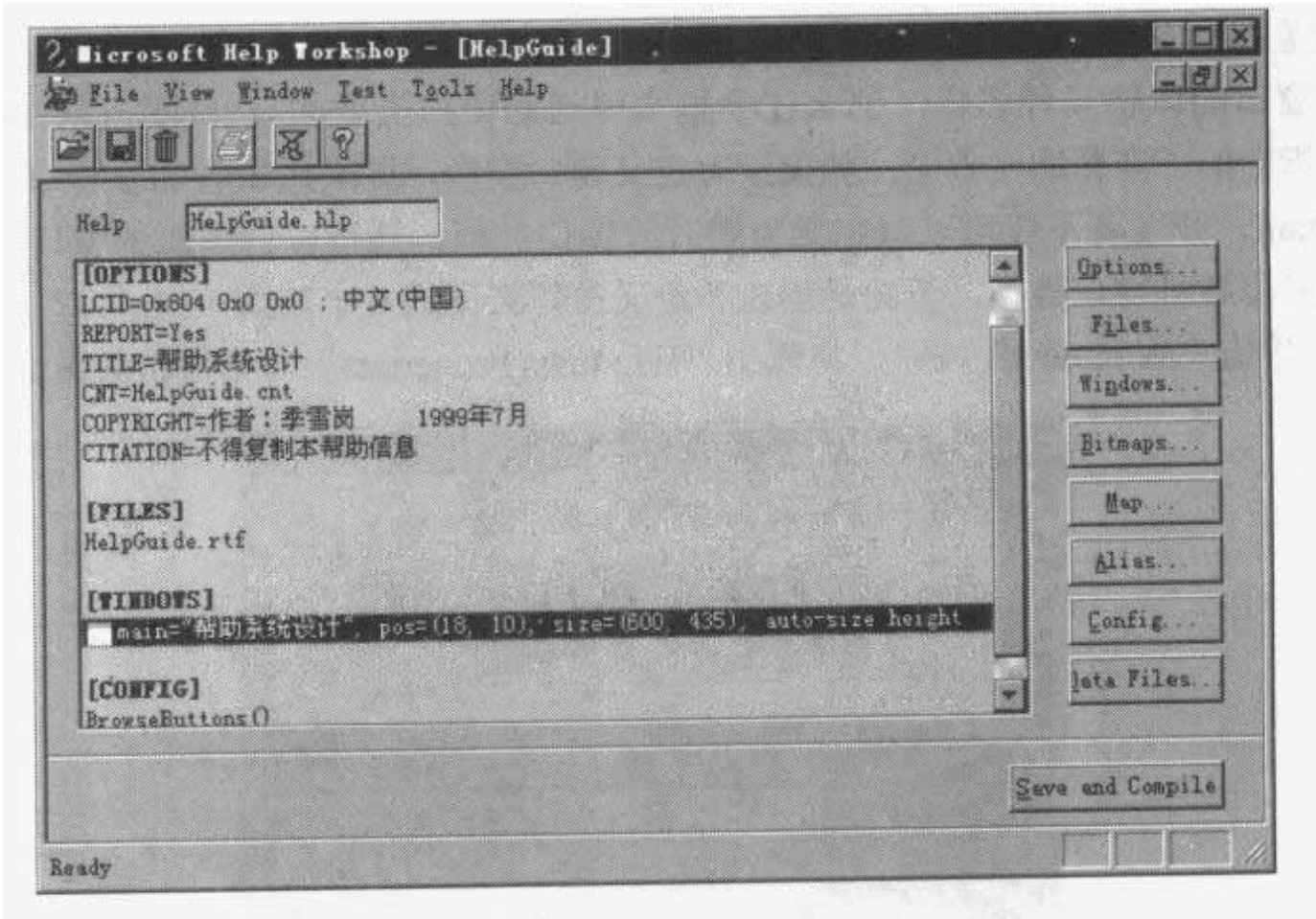


图 21-7　编辑项目帮助文件

首先单击“Files”按钮，然后在“Topic Files”对话框中添加帮助主题文件。接下来，单击“Options”按钮，在弹出的“Options”对话框中设置各选项，如图 21-8 所示。因为 HelpGuid 帮助文件中用到了目录文件，所以需要在“Files”选项卡中指定帮助目录文件的名称。同时，在该选项卡中还可以指定待创建的帮助文件的名称以及主题文件的名称等。其他几个选项卡

都分别用来设置另外一些信息，如“General”选项卡可设置版权信息；“Compression”选项卡用来选择压缩方式“Sort”选项卡用来设定帮助文件采用的语言和关键词排序方式；“Build Tags”选项卡通过使用主题中的连编标记，设定哪些主题包含在连编内，哪些主题排除在连编之外。

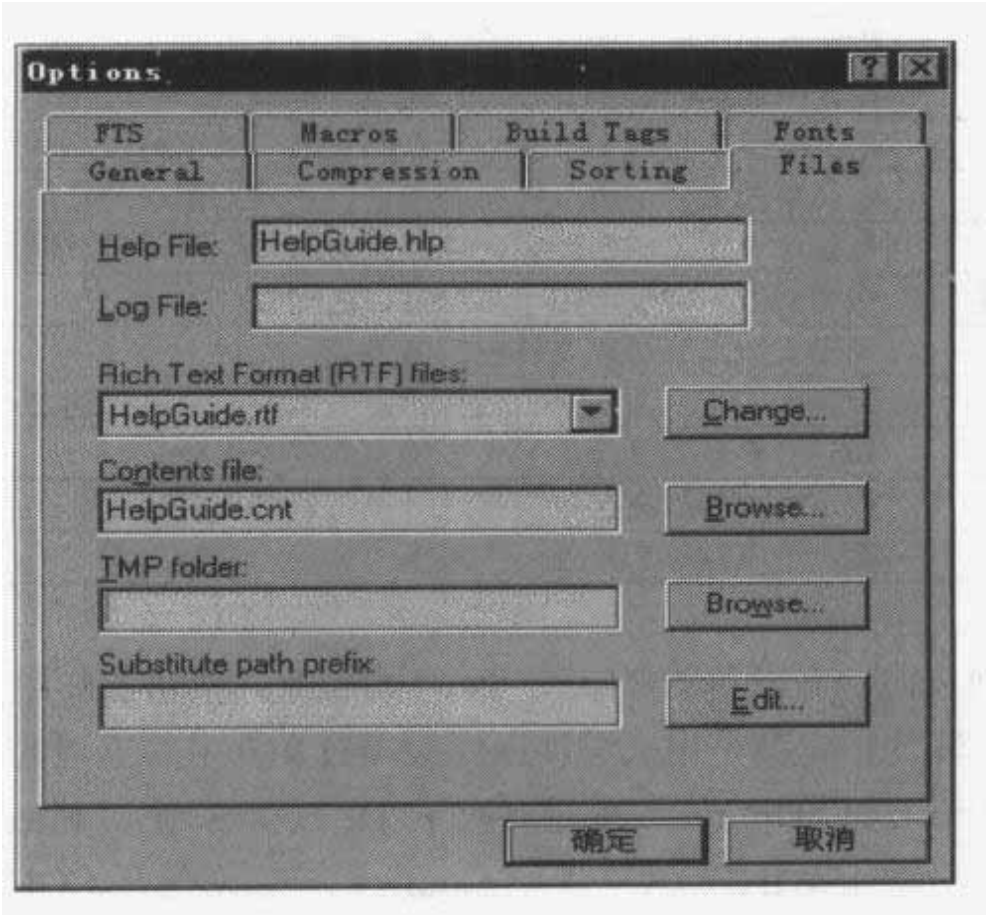


图 21-8　设置帮助项目文件中【options】节的选项

4. 设置窗口类型

帮助窗口的风格多种多样，那么在哪里实现帮助窗口类型的设计呢？同样是在帮助项目文件中指定的。在帮助文件中，如果没有定义窗口类型，则采用缺省的主窗口类型，其类型名为 main。开发者不仅可以自定义为现有的窗口类型，还可以创建新的窗口类型。辅助窗口可以采用各种窗口类型。下面是创建和定义窗口类型的几个步骤：

（1）单击“Windows”按钮，将弹出“Windows Properties”对话框，如图 21-9 所示；

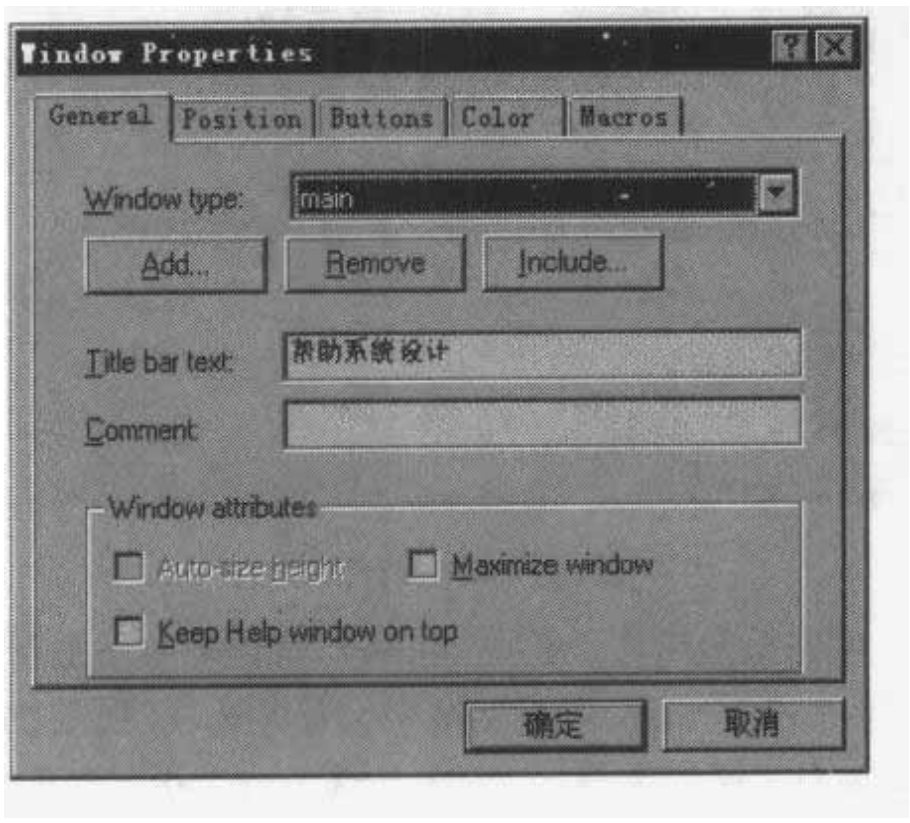


图 21-9　定义窗口类型

（2）在“General”选项卡中单击“Add”按钮，然后在“Add a New Window Type”对话框中输入窗口类型名称；

（3）在“Position”选项卡中指定该类窗口的按钮栏中的按钮，例如，要想用浏览按钮，那么应选中“Browse”选项。

（4）在“Color”选项卡中分别设定窗口主题区域和非滚动区域（包括主题标题）所采用的背景色。

（5）在“Macros”选项卡中添加并编辑在打开该类窗口时将自动运行的宏；

（6）单击“OK”按钮，退出“Windows Properties”对话框，然后再单击“Config”按钮，接着在“Configuration Macros”对话框中加入一些必要的宏来生成菜单和菜单项。

当定义了窗口类型后，就可以在帮助目录文件中的主题目录项中加上需要的窗口类型。



调用帮助文件



遇到难题

这是进行帮助系统设计的最后一步。也是将应用程序与帮助文件结合在一起的一步。



钥匙在此

帮助函数 WinHelp

WinHelp 是标准的 Window API 函数，应用程序通过调用该函数启动 WinHelp.exe 程序来打开帮助文件，并向它传递必要的参数，使帮助文件以指定的方式显示帮助信息。

在应用程序中要实现联机帮助，应明确两个过程和方式：首先是帮助请求方式，如通过菜单、按钮、快捷键或鼠标就地单击来请求帮助；其次是帮助信息的显示方式。无论哪一个过程或者方式，在应用程序中最终都是由 WinHelp 函数来实现的。

WinHelp 函数的原型说明如下：

```
BOOL WinHelp（
    HWND hWndMain,      // 请求帮助的视窗句柄
    LPCTSTR lpszHelp,   // 帮助文件的路径名
    UINT uCommand, // 帮助类型
    DWORD dwData   // 附加的信息
  ）;
```

参数的含义如下：

1. hWndMain

指定请求帮助的视窗句柄。WinHelp 函数使用这个句柄来追踪是哪个应用程序在请求帮助。该参数在 Delphi 中的一般方式是“视窗名.Handle”。

2. lpszHelp

指定帮助文件的路径名，包括帮助文件的所在路径（如果必要的话）以及帮助文件名。如果主题显示在次窗口中（secondary window）而不是主窗口中，该文件名后会跟有一个尖括号“>”以及次窗口的名称。次窗口应该在帮助项目文件（.hpj）的[WINDOWS]部分进行定义。

3. uCommand

指定帮助请求的类型。可能的类型值以及它们对 dwData 参数的影响，请参看表 21-6 所示。

4. dwData

指定附加数据。使用的参数值取决于 uCommand 参数。

表 21-6　　帮助请求的主要类型

uCommand	动作	dwData
HELP_COMMAND	执行帮助宏	指定帮助宏的字符串的地址
HELP_CONTEXT	显示由.hpj 文件[MAP]部分的上下文标识符所指定的主题	即主题标识号（在帮助项目文件的[MAP]部分中定义的），为无符号长整数
HELP_CONTEXTPOPUP	以弹出式窗口显示由.hpj 文件[MAP]部分的上下文标识符所指定的主题	同上
HELP_HELPONHELP	显示如何使用 WinHelp 的帮助信息	被忽略，设为 0
HELP_KEY	根据关键词来查找主题。如果完全匹配，则在关键词表中显示该主题。如果匹配多条，则会显示“找到的主题”对话框，列出所有匹配主题的标题，让用户进行选择	关键词（字符串）的地址
HELP_MULTIKEY	显示由关键词在选择性关键词表（an alternative keyword table）中指定的主题	MULTIKEYHELP 结构变量的地址
HELP_PARTIALKEY	根据关键词来查找主题。如果完全匹配，则在关键词表中显示该主题。如果匹配多条，则会显示【索引选项卡】	关键词（字符串）的地址
HELP_QUIT	通知 WinHelp.exe 程序退出。如果没有其他应用程序在使用帮助的话，操作系统将关闭 WinHelp.exe 程序	被忽略，设为 0
HELP_FINDER	显示帮助主题对话框	被忽略，设为 0
HELP_SETCONTENTS	指定目录主题。当用户单击“目录”按钮时，帮助应用程序将显示该目录主题	即目录主题标识号，为无符号整数
HELP_SETINDEX	指定被显示在帮助主题对话框索引中的主题词表	即索引主题标识号，为无符号整数
HELP_SETWINPOS	如果帮助视窗被极小化或存在内存中，则可以设置其大小及位置	HELPWININFO 结构变量的地址



跟我来

WinHelp 函数主要使用的就是 uCommand 和 dwData 参数，它们的组合使用可以控制如何调用帮助信息。现就几种常用方法给以示例。

1. 显示帮助主题对话框

访问帮助信息最常用的一种方式是通过菜单命令来打开帮助主题对话框。而应用程序是通过调用 WinHelp 函数，使用 HELP_FINDER 命令参数来打开指定帮助文件的帮助主题对话框。

下面这段程序代码是处理应用程序 Form1 的【帮助】菜单中的菜单命令的示例：

```
procedure TForm1.HelpClick（Sender: TObject）;      //菜单的单击事件
begin
        if （not winhelp（Handle,'HelpGuide.hlp',HELP_FINDER,0））then begin
            showmessage（'不能访问帮助信息！'）;
            exit;
        end;
end;
```

2. 直接显示帮助信息主题

在使用 WinHelp 函数时，只需把其中的命令参数设为 HELP_CONTEXT，应用程序就可以根据给定的主题标识号来显示特定的帮助主题。

在应用程序中，可以根据应用程序的上下文环境直接获得其特定的帮助信息，如在某一环境下直接按【F1】就可以获得相应的帮助主题。

例如，有一个编辑框，帮助文件中为它提供了主题标识符为 HELPFILE 的帮助主题。要直接显示这个主题，首先应在帮助项目文件中的 MAP 节中加上如下一行：

```
[MAP]
HELPFILE=8888
```

即把主题标识符 HELPFILE 映射成为主题标识号 8888。然后，为应用程序指定帮助文件的文件名。可以设定窗体 Form.HelpFile 属性，也可以赋给 Application.HelpFile，本例为 HelpGuide。最后，设定编辑框的 HelpContext 的属性值为 8888。这样在应用程序执行时，当光标聚焦在该编辑框上时按【F1】即可直接显示标识符为 HELPFILE 的帮助主题。虽然我们并不直接调用 WinHelp 函数，但其底层用的还是 WinHelp。上述在应用程序中设置属性的操作与以下代码是等价的：

```
procedure TForm1.Edit1KeyDown（Sender: TObject;
var Key: Word; Shift: TShiftState）;//编辑框的按键事件
begin
    if key=VK_F1 then        //如果按的是【F1】键的话
        winhelp（edit1.handle,'HelpGuide',Help_context,8888）;
end;
```

3. 通过关键词搜索并显示帮助信息主题

应用程序显示帮助主题的另一种途径可以是通过搜索帮助主题关键词来实现。带帮助命令参数 HELP_KEY、HELP_PARTIALKEY 的 WinHelp 函数都可以完成这项工作，主题分别基于整个或部分关键词。下面的代码就是使用 HELP_KEY 命令的一个典型例子：

```
var
    s:string;
begin
    s:='帮助文件';
    if （not winhelp（Handle,'HelpGuide.hlp',HELP_KEY,DWORD（@s））） then begin
            showmessage（'不能访问帮助信息！'）;
            exit;
    end;
```



在缺省方式下，可以被搜索的主题关键词只能是由 K 脚注定义的。如果要使用其他关键词，那就需要定义关键词表，然后再使用带 HELP_MULTIKEY 命令参数的 WinHelp 函数。

4. 显示弹出式帮助主题窗口

如果要显示具体的控件或菜单的帮助信息，应用程序可以调用带 HELP_WM_HELP 或 HELP_CONTEXTMENU 命令参数的 WinHelp 函数，以显示弹出式的帮助主题窗口。如何为控件增加在线帮助，请参考 Delphi4.0 或 Delphi5.0 的“creating custom componets”主题目录。

第二十二章

安装盘制作

热点透视

假如应用程序是用于商业目的的话，肯定不要忘了为应用系统和手册等等制作安装盘并进行打包。本章主要内容：

- 制作安装盘



制作安装盘



遇到难题

假如应用程序是用于商业目的的话，肯定不要忘了为应用系统和手册等等制作安装盘并进行打包。



钥匙在此

目前在软件安装的软件中使用最多的是 Install Shield，世界上许多著名的软件厂商均采用 Install Shield 包装他们所开发的软件，它的使用非常简单。

在 Delphi 的 c/s 版本中一般都提供了 InstallShield Express For Delphi 的简化版。本章节使用的版本是 Delphi 4.0 c/s 版本中的 Install Shield Express，它特别为了 Delphi 作了一些修改，本章将以 Delphi 开发出来的 Windows 95/98/NT 的应用程序为例介绍怎么制作安装盘。



跟我来

下面我们以 Delphi 内带的 Fishfact 程序为例，示范如何用 Install Shield 来自动安装软件。Fishfact 这个程序包括以下两个部分：

1. 执行程序

Fishfact.exe

2. 数据库相关文件

Biolife.db

Biolift.mb

Biolife.px

1. 启动 Install Shield Express

介绍完了 Fishfact 程序所包含的文件后，就可以启动 Install Shield 安装程序。安装步骤如下：

（1）启动时 Install Shield 会询问您下面的选项（如图 22-1），因为我们要产生新的 Setup Project 因此选择“Create a new setup Project”。

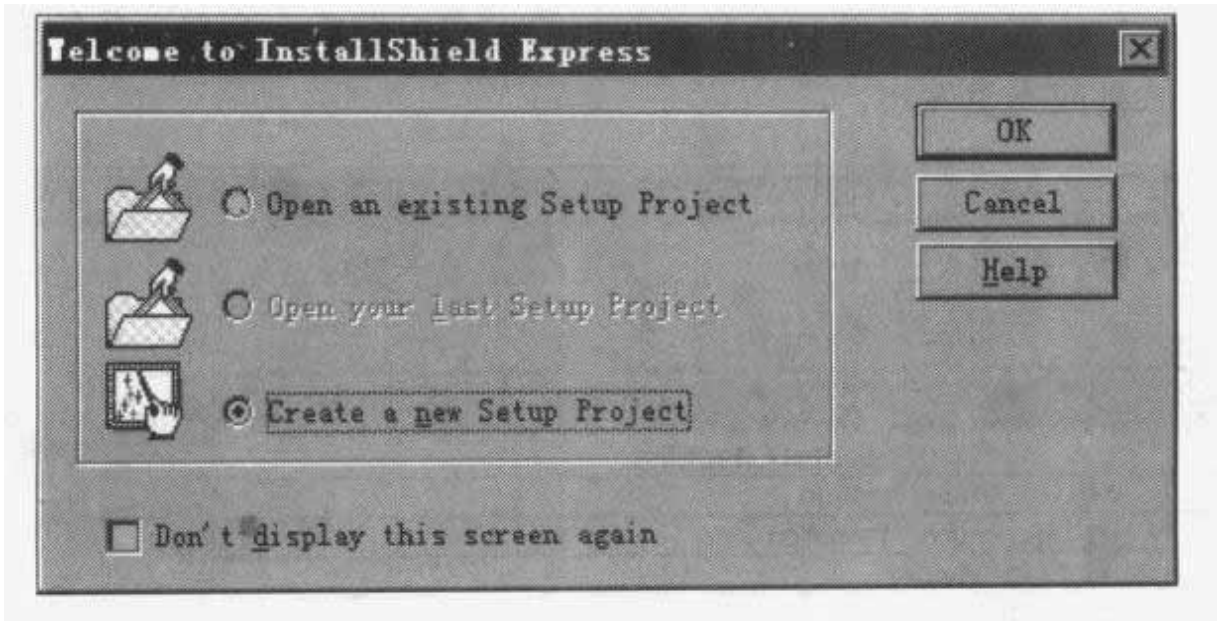


图 22-1　建立新工程（project）的选项视窗

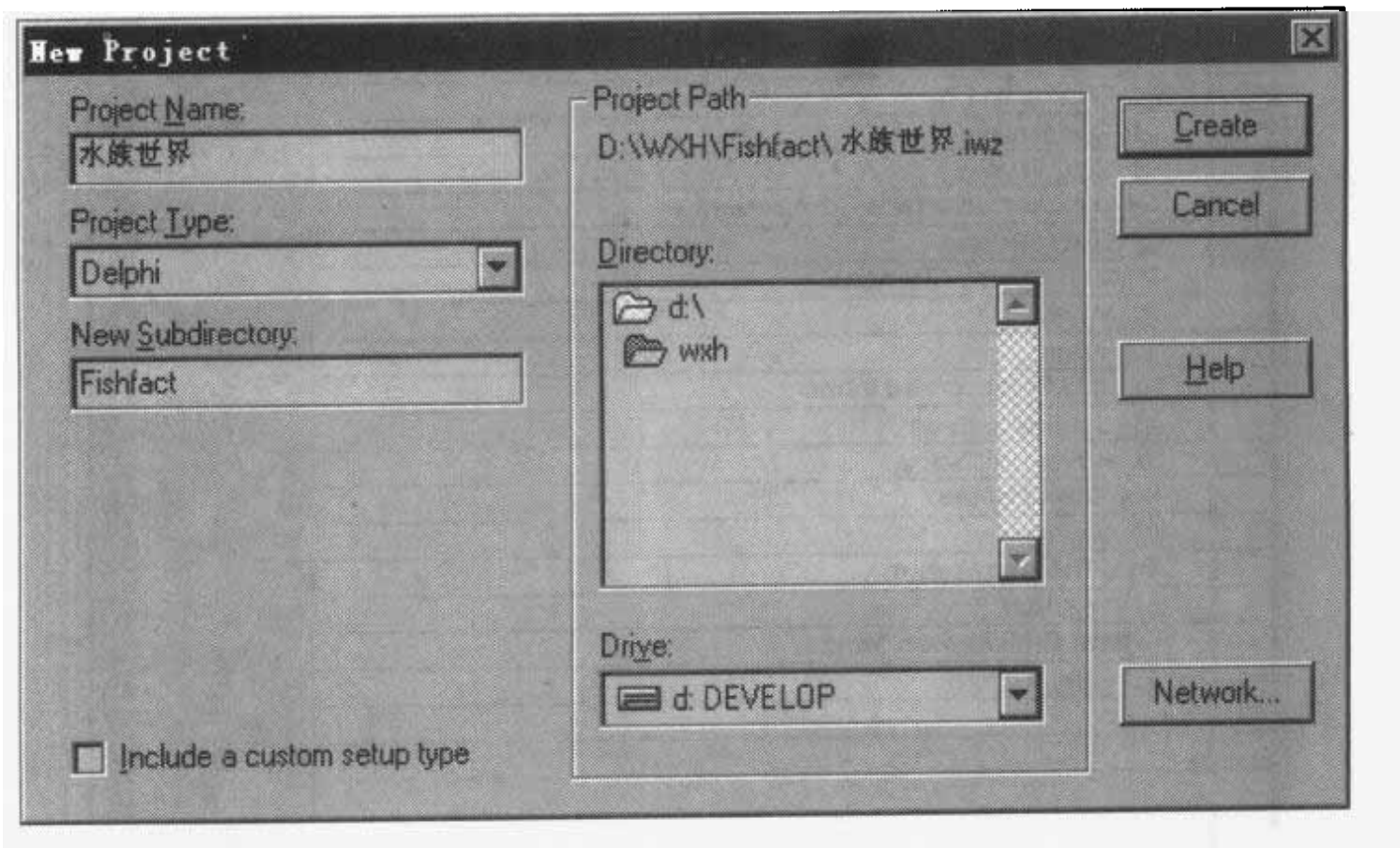


图 22-2　“水族世界”工程的参数设定

（2）新工程的参数说明：

① Project name（工程名称）：为软件的中文名称；

② Project Type（工程类型）：由于这个版本是专为 Delphi 设计的，所以只有 Delphi 的选项；

③ New Subdirectory（新的子目录）：安装所需相关文档或者文件都放在了该目录下；

④ Include a custom setup type （包含一个自定义安装类型）：一般较大型的软件安装时有下列数个选项：

- Typical
- Compact
- Custom or Complete

若需要自定义安装时，请选择此选项；本范例未使用自定义安装故省略此部份的安装说明。

（3）填好了以上的基本内容后，请按“Create”按钮，Install Shield 会出现“Setup checklist（程序检核清单）”的视窗（如图 22-3），依照视窗的先后次序，按步就班的填完每一个视

窗要求您提供的参数或选项，系统就会自动将相关内容包装成一个自动化的安装程序，这些程序就可以通过软盘或光盘发布。

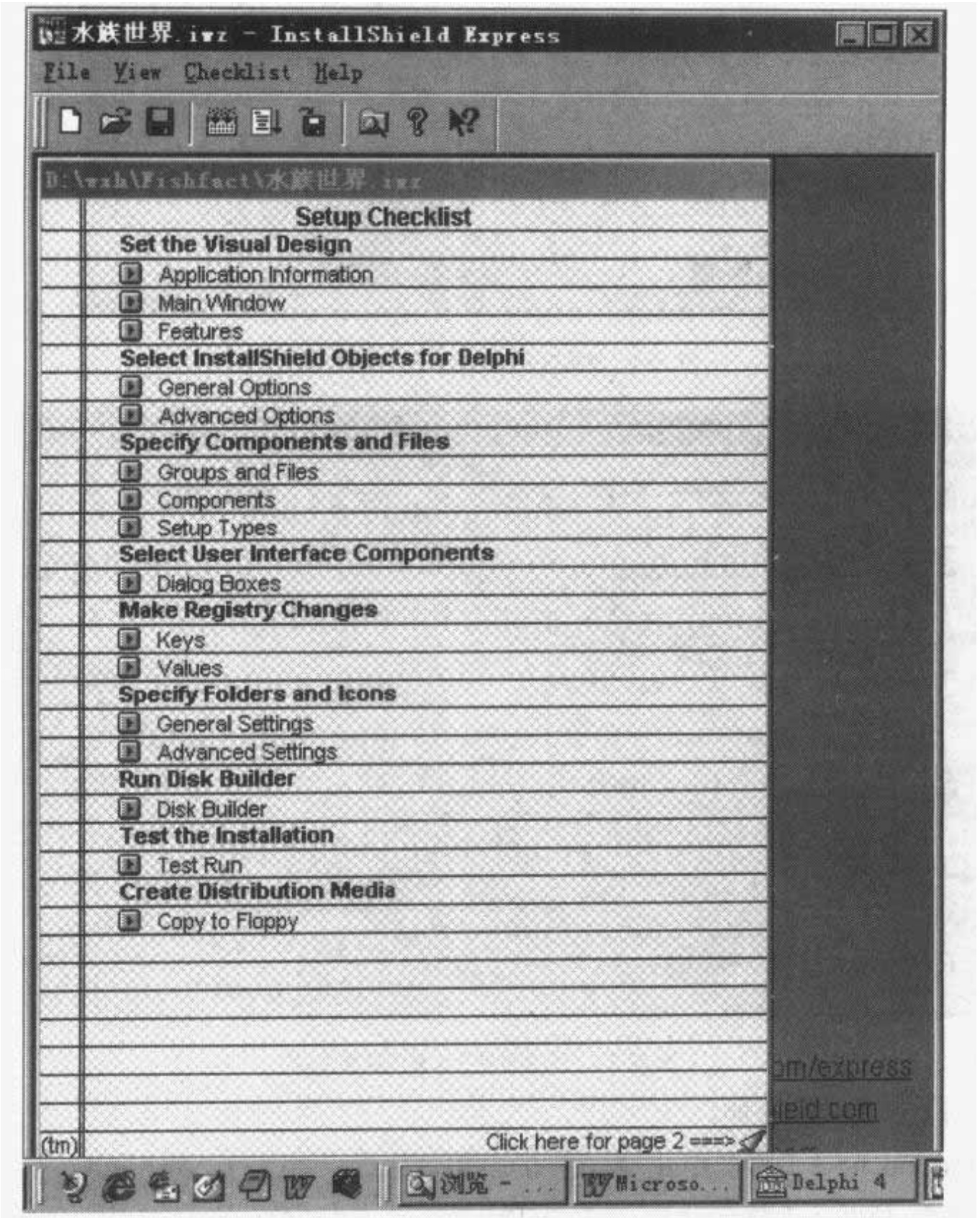


图 22-3　程序检核清单

程序检核清单分为六大设定项目，分别为：

① Set the Visual Design（设定视觉化设计）

② Select InstallShield Objects for Delphi（选择 InstallShield 的 Delphi 物件）

③ Select components and files（指定元件和文件）

④ Select User Interface Components（选择使用者界面元件）

⑤ Make Registry Changes（变更注册表）

⑥ Specify Folders and Icons（指定文件夹及图标）

2. 设定步骤

（1）设定视觉化设计

① App info（应用程序信息）（如图 22-4 所示）

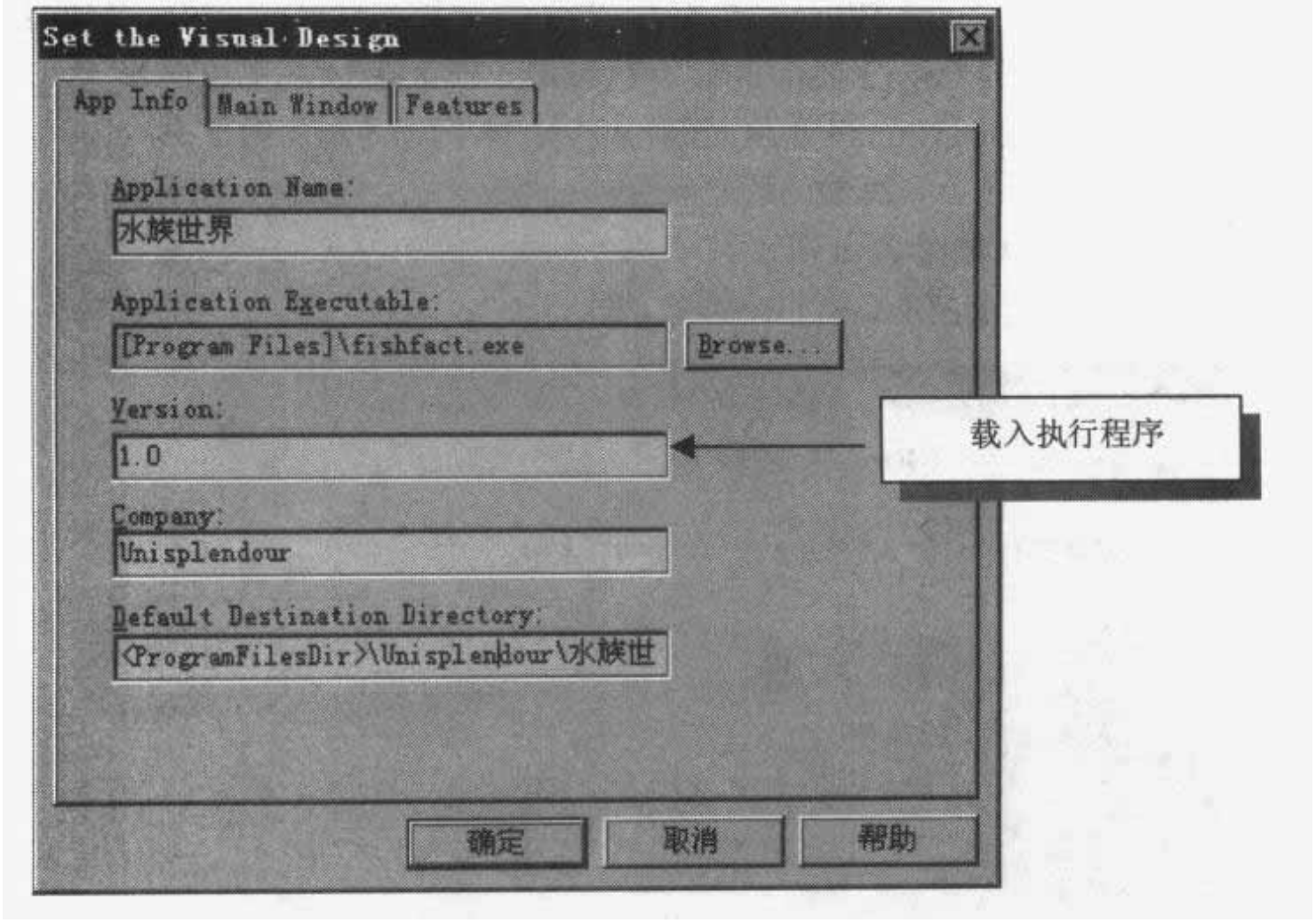


图 22-4　应用程序信息

Application Name（应用程序名称）：　程序的系统名称；

Application Executable（可执行程序）：要包含的可执行程序名称（含路径）；

Version（版本）：　1.0；

Company（公司）：开发此软件的公司（Crystal Studio）；

Default Destination Directory（缺省的安装目录）：安装时的预设目录。（内定为<ProgramFilesDir>\公司\应用程序名称）

② Main Window（主视窗）（如图 22-5 所示）

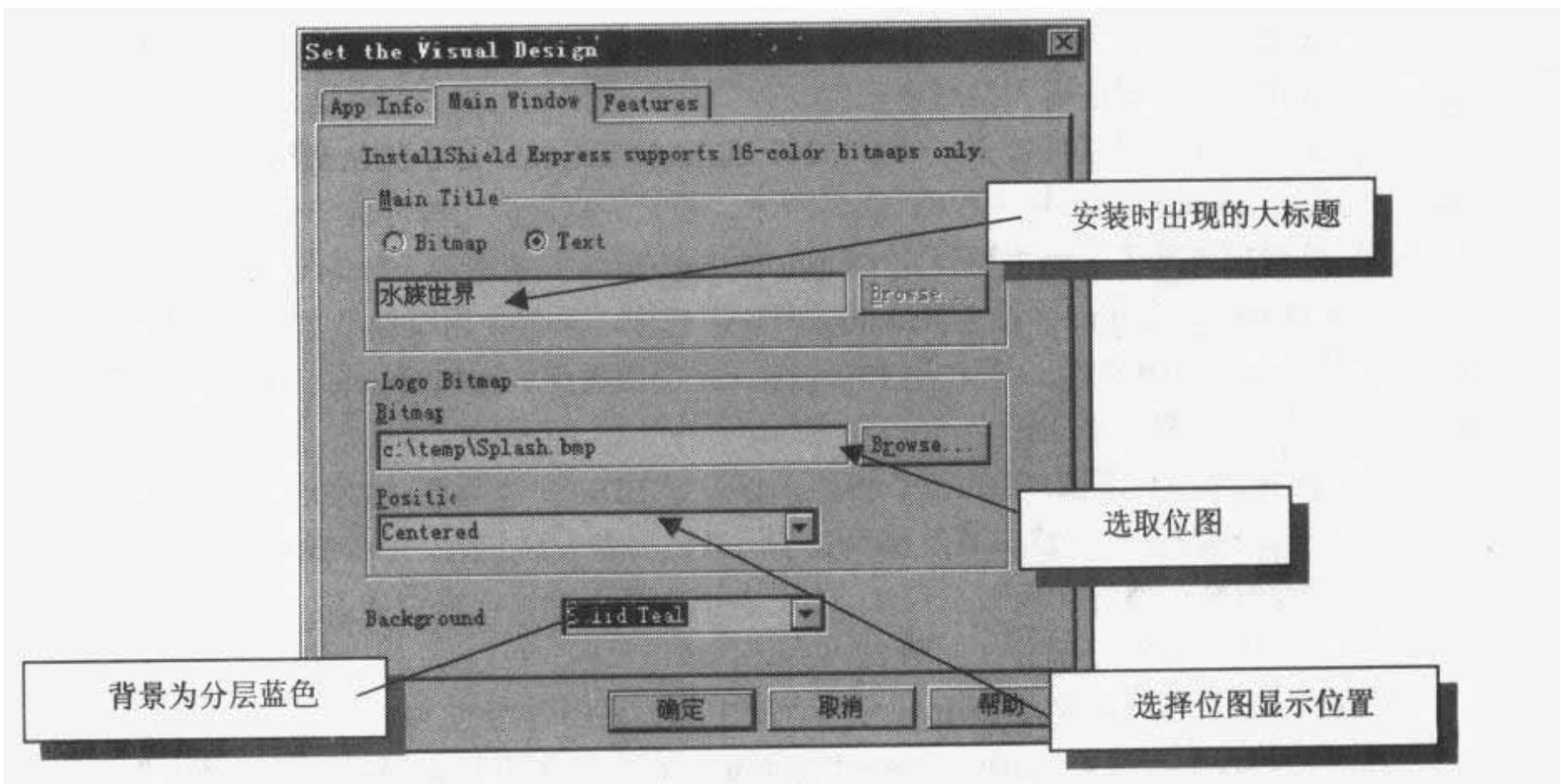


图 22-5　主视窗

Main title（主标题）： 可选 Bitmap 或文字，执行安装程序时显示的第一个画面的标题。
Logo Bitmap： 软件的 Logo 图形，可以自己设定（只支持 16 色的位图）；
Position（位置）： Logo Bitmap 在画面中的位置，本例选为画面的中央；
Background（背景颜色）： 画面的背景颜色。
③ Features（功能）（如图 22-6 所示）
自动卸载。若选择此项表示具有自动卸载功能。

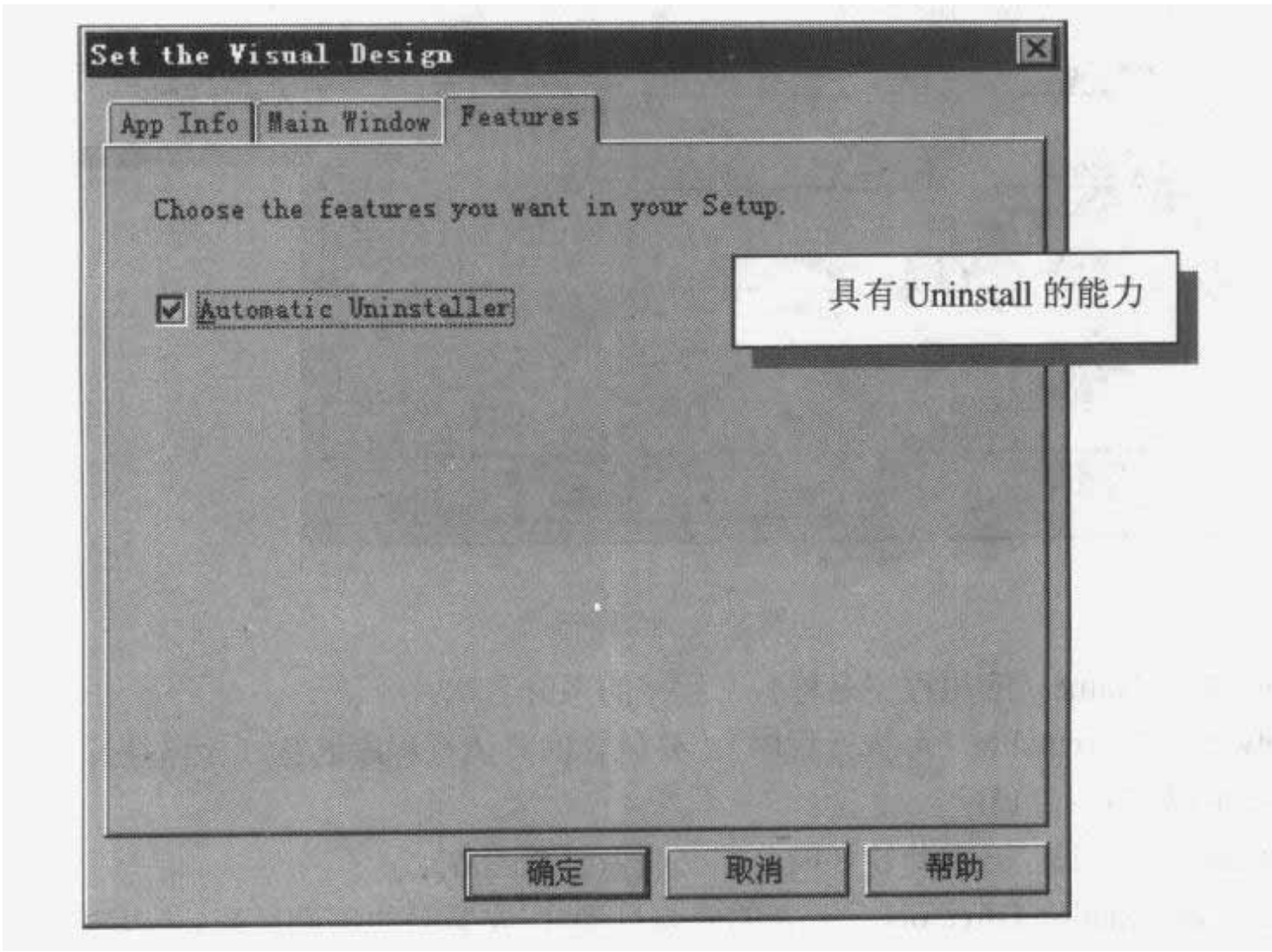


图 22-6 功能特性

（2）选择 InstallShield 的 Delphi 物件

在可视化或面向对象环境中开发的应用程序一般使用了额外的 DLL 或其他文件，为了使用这些文件一般需要注册入口或者配置文件。动态选择应用程序所必要的文件或系统配置对于制作安装程序来说是件较困难的事，InstallShield Express 为了能包括这些文件提供了较容易的方法：InstallShield 物件如图 22-7 所示。若是数据库应用程序应该包括数据库的安装选项，这些选项包括了一些主要设定 Delphi 的 Database 安装选项，Delphi 数据库是由 BDE（Borland Database Engine）及 SQL Links 两项所组成，一般使用桌面型数据库，如 dBase、Paradox 等，选用 BDE 即可，若使用 Client/Server 的数据库，例如 Oracle、Informix、Sybase、Interbase 等则两者都需要选择。

选取 BDE 选项后请按“Setting”按钮，以设定 BDE 安装类型（如图 22-8 所示）。

BDE 安装类型分为完整安装及部份安装两种，完整安装包含 BDE 所有数据库驱动程序及 SQL 查询引擎；部份安装则不包含 SQL 查询引擎。本项选择完毕请按“下一步”，接下來就是 BDE Alias 的设定部分（如图 22-9 所示）。

请按“New”按钮，新增 Delphi 应用程序中数据库所使用的 Alias，如图 22-10 所示。

Alias 名称视程序中 DatabaseName 的值而定，在本例中为 FishDB 然后按“OK”按钮，将现现如图 22-11 所示的窗口。请继续按“下一步”按钮将出现如图 22-12 的窗口。



图 22-7 InstallShield 的 Delphi 物件

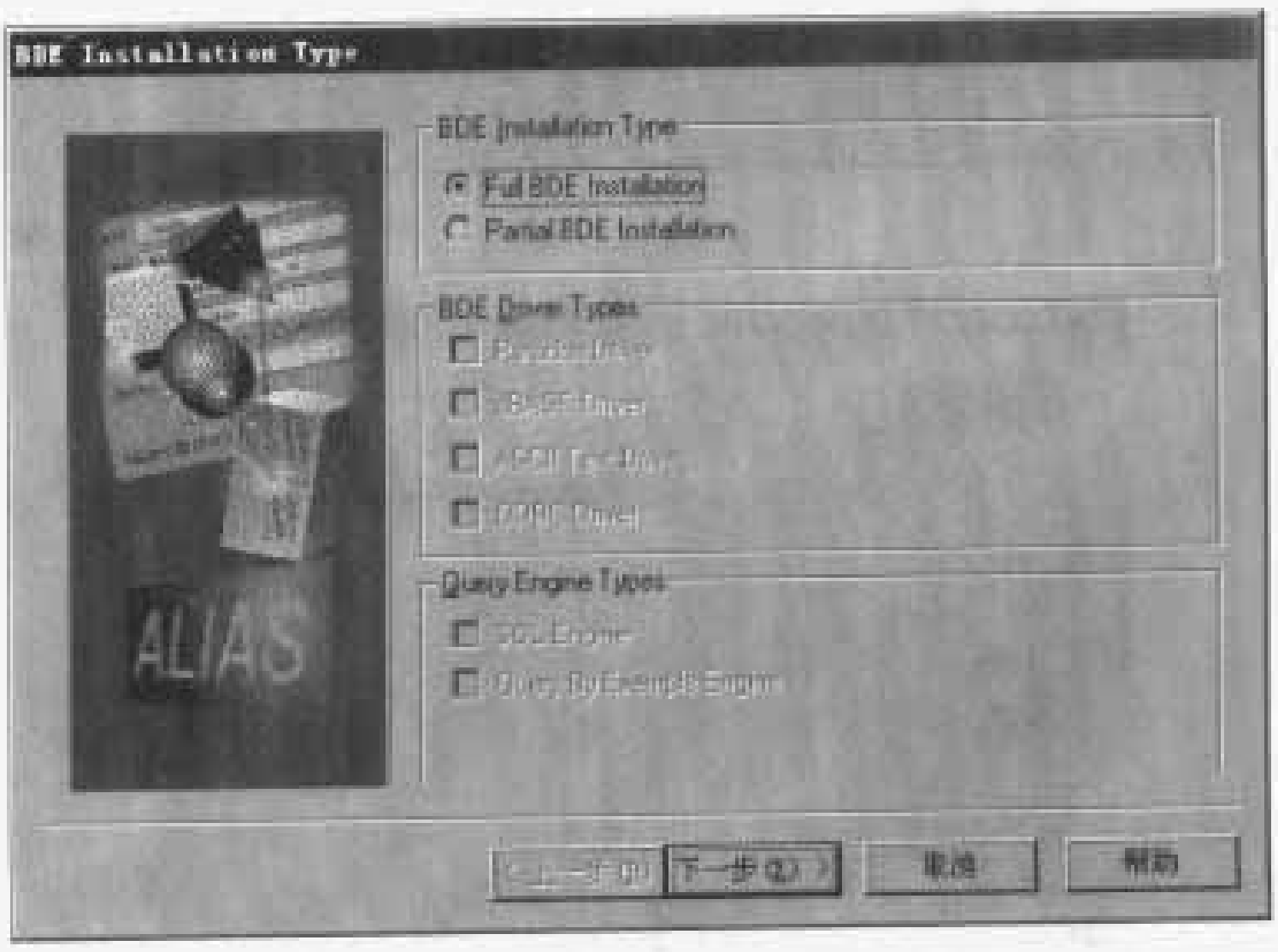


图 22-8 BDE 安装类型

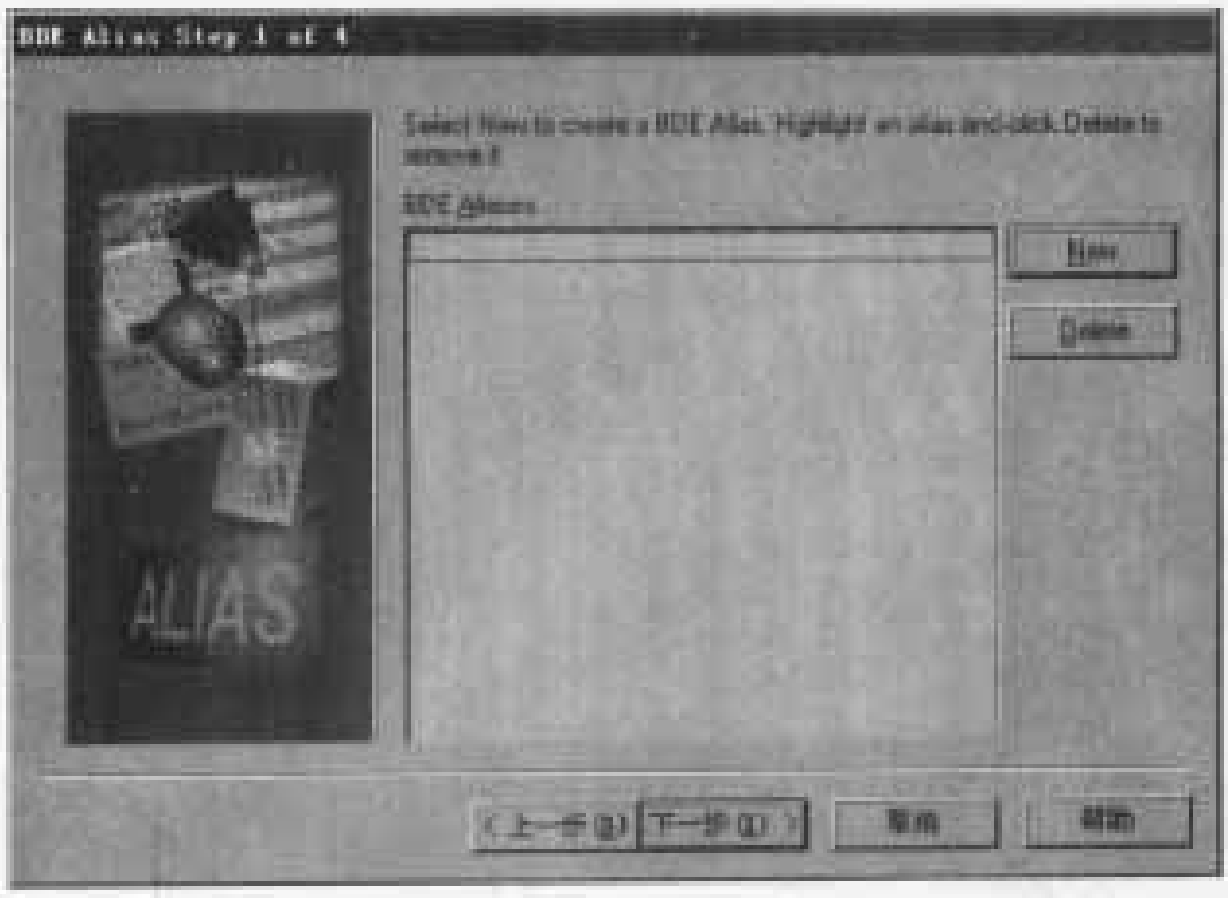


图 22-9 BDE Alias 步骤 1 of 4

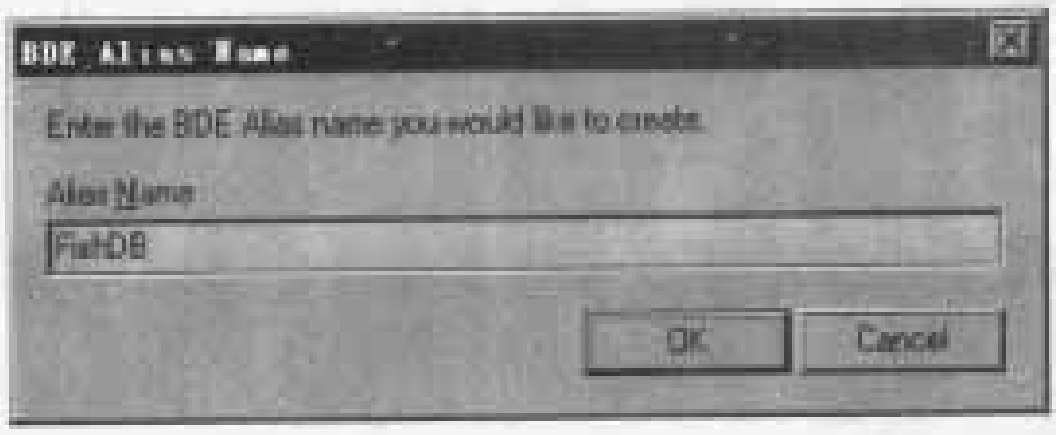


图 22-10 BDE Alias

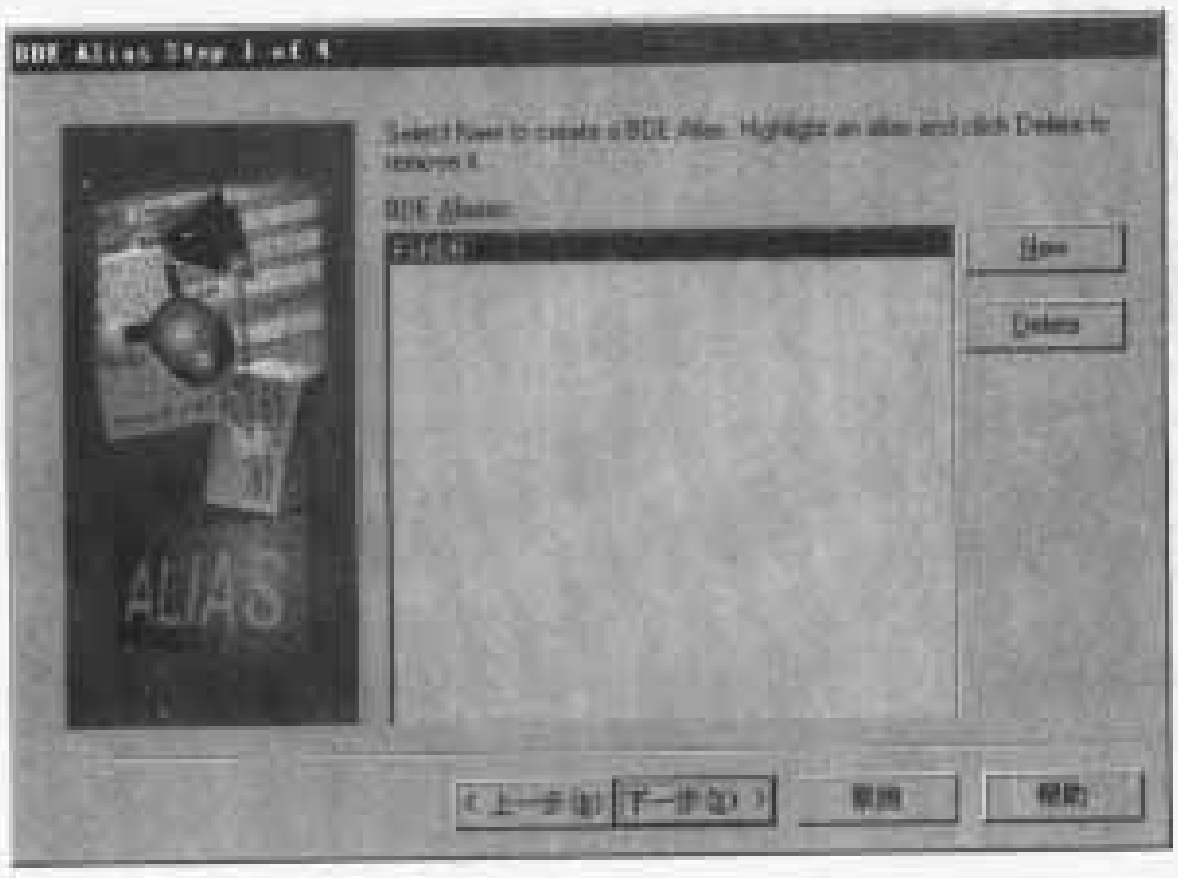


图 22-11 选取 BDE Alias 后的视窗

此处决定 BDE 的配置文件 IDAPI.cfg 是存成 16 位还是 32 位类型，本例以 Windows 95/NT 为例，故不选。按“下一步”按钮，然后会出现如图 22-13 的窗口。

若应用程序用到许多的 Alias，请依次添加相应的 Alias，设定完毕后请按“下一步”按钮。

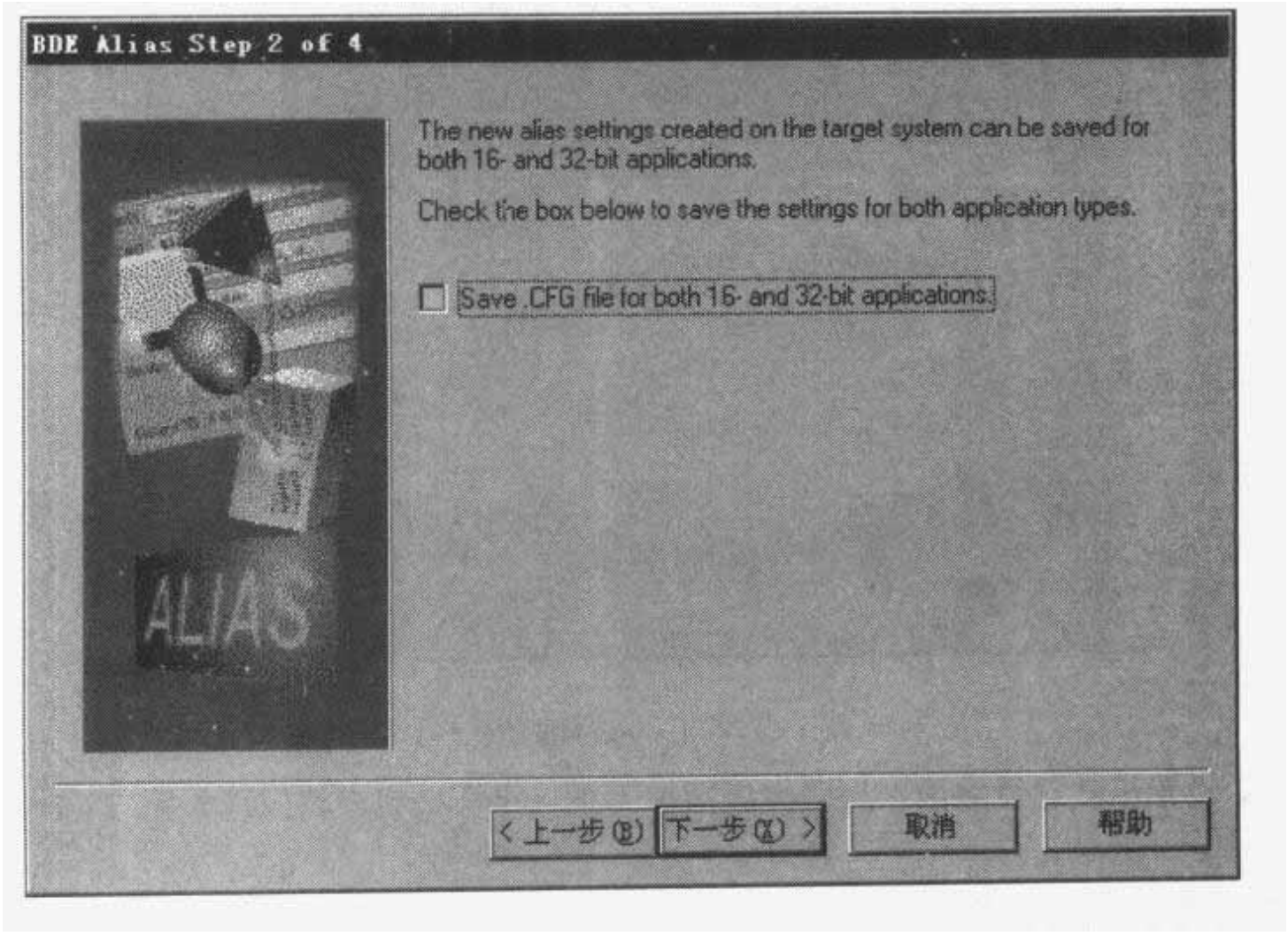


图 22-12　配置文件设置

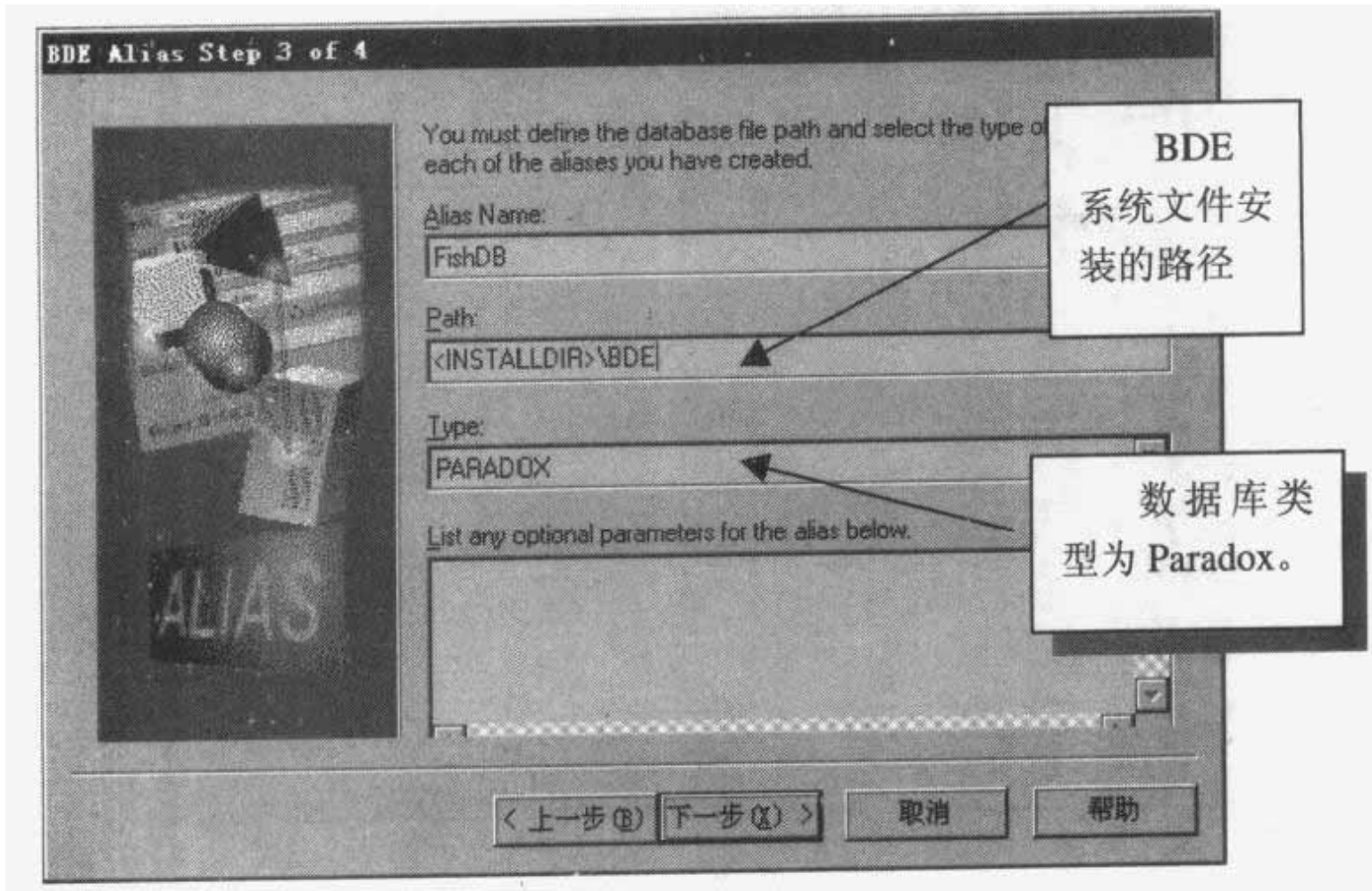


图 22-13　设定 Alias 参数

图 22-13 中“Path”为 BDE 系统文件打算要安装的目录。考虑到用户的目标系统各不

相同，InstallShield Express 包含了一组特殊目录名，这些目录代表了安装时的一组信息，如<InstallDir>、<WinDir>等，本例为“<InstallDir>\BDE”,表示安装在用户指定目录的 BDE 子目录下。

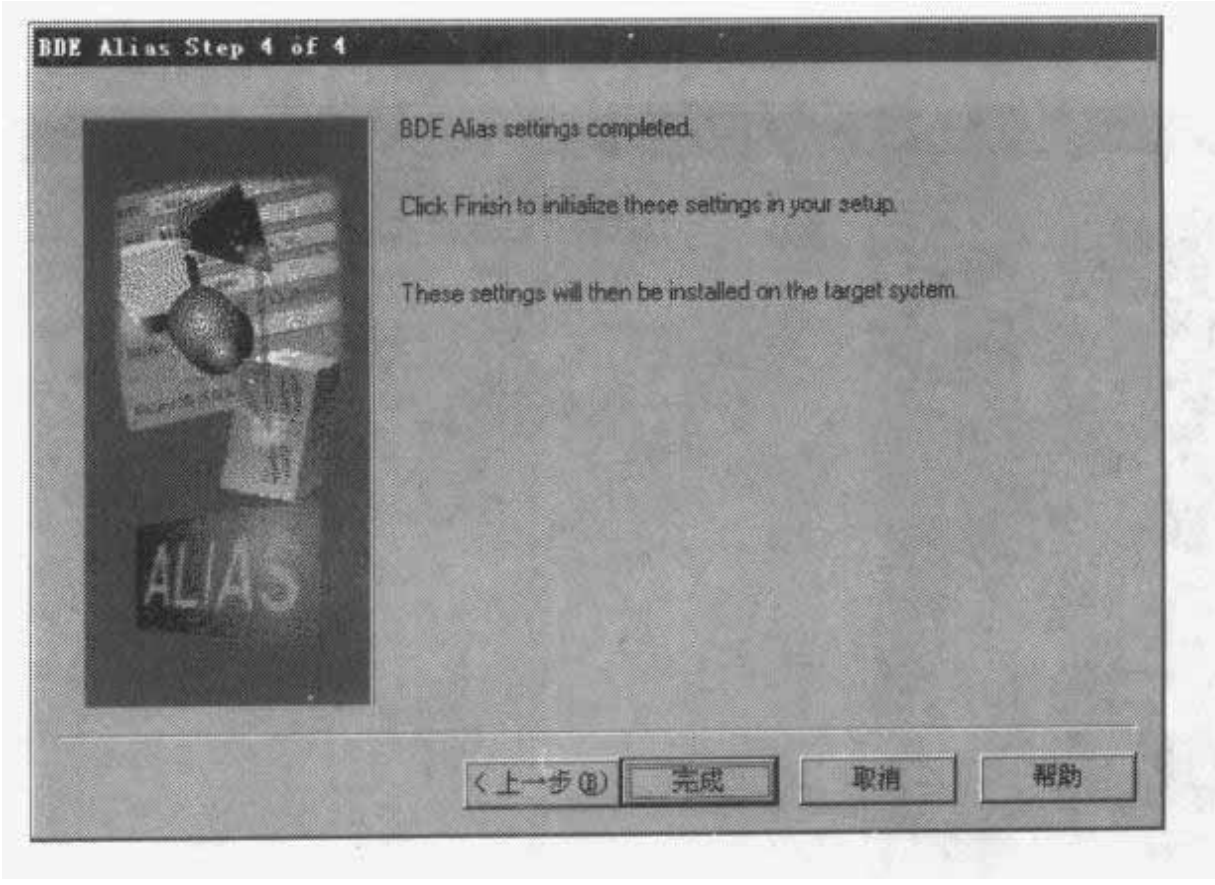


图 22-14　Alias 设置完成

Alias 设置至此已经完成。设置完成后按“下一步”按钮，将出现如图 22-14 所示的窗口。

（3）指定组件和文件

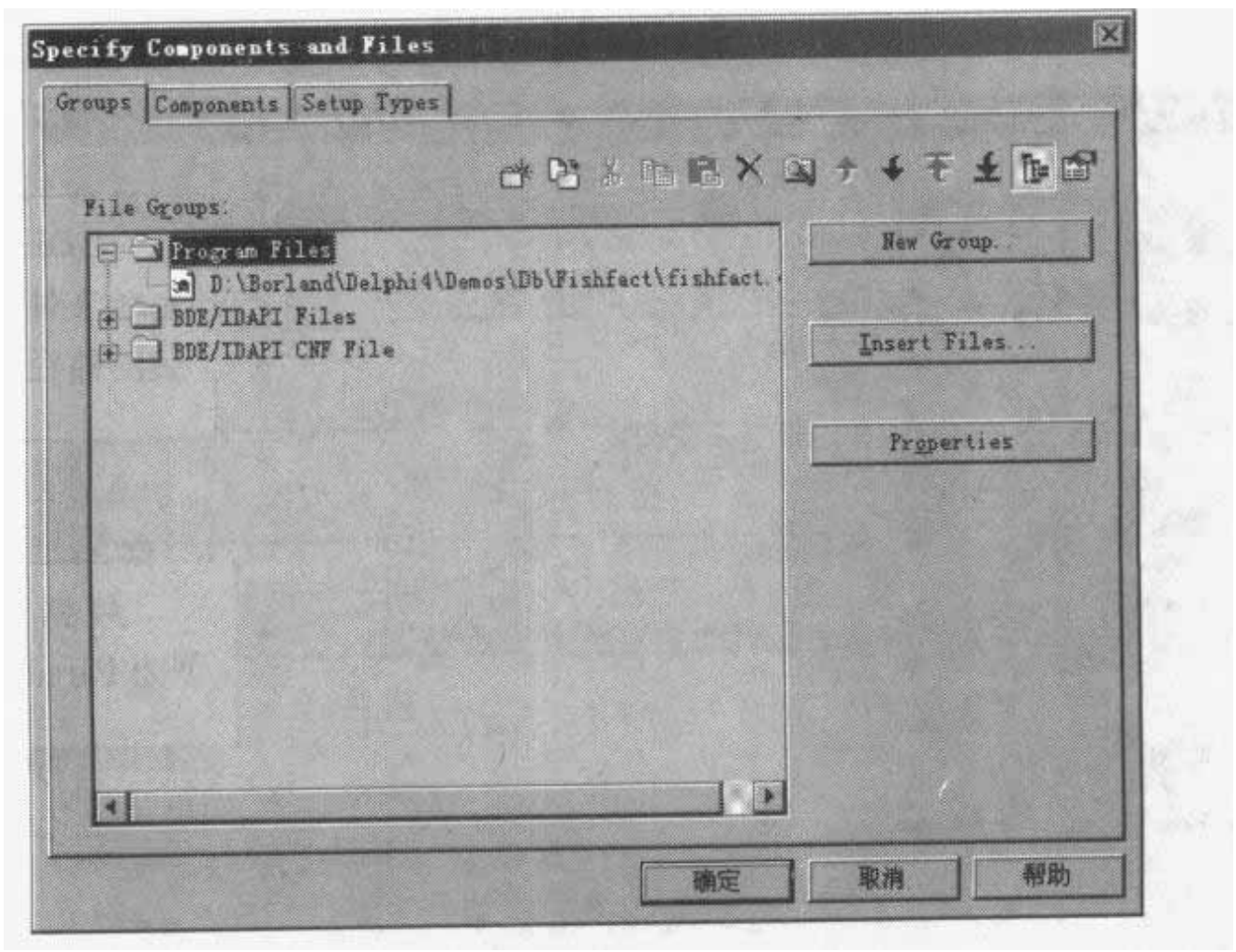


图 22-15　指定组件和文件

文件群组（File Groups）：是文件的集合，如图 22-15 所示，本例中的 Program Files 为执行程序集合。其他三个群组由系统自动产生，里面包含了所需的数据库驱动程序。

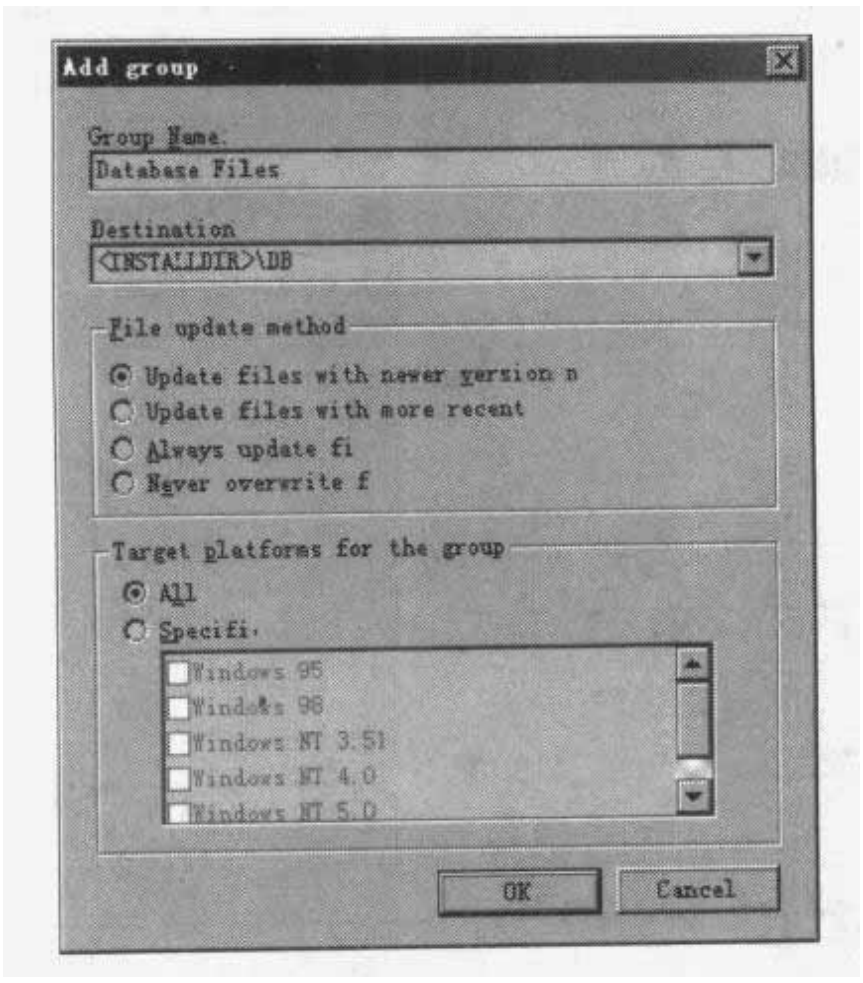


图 22-16　加入群组

在本例中若将数据库相关文件放在“Database Files”群组中，请点击“New Group”，并输入群组名称“Database Files”（如图 22-16）。

然后点击“Launch Explorer”,启动资源管理器，将选中的数据库文件拖（Drag-Drop）至“Database Files”群组中放下即可如图 22-17 所示。

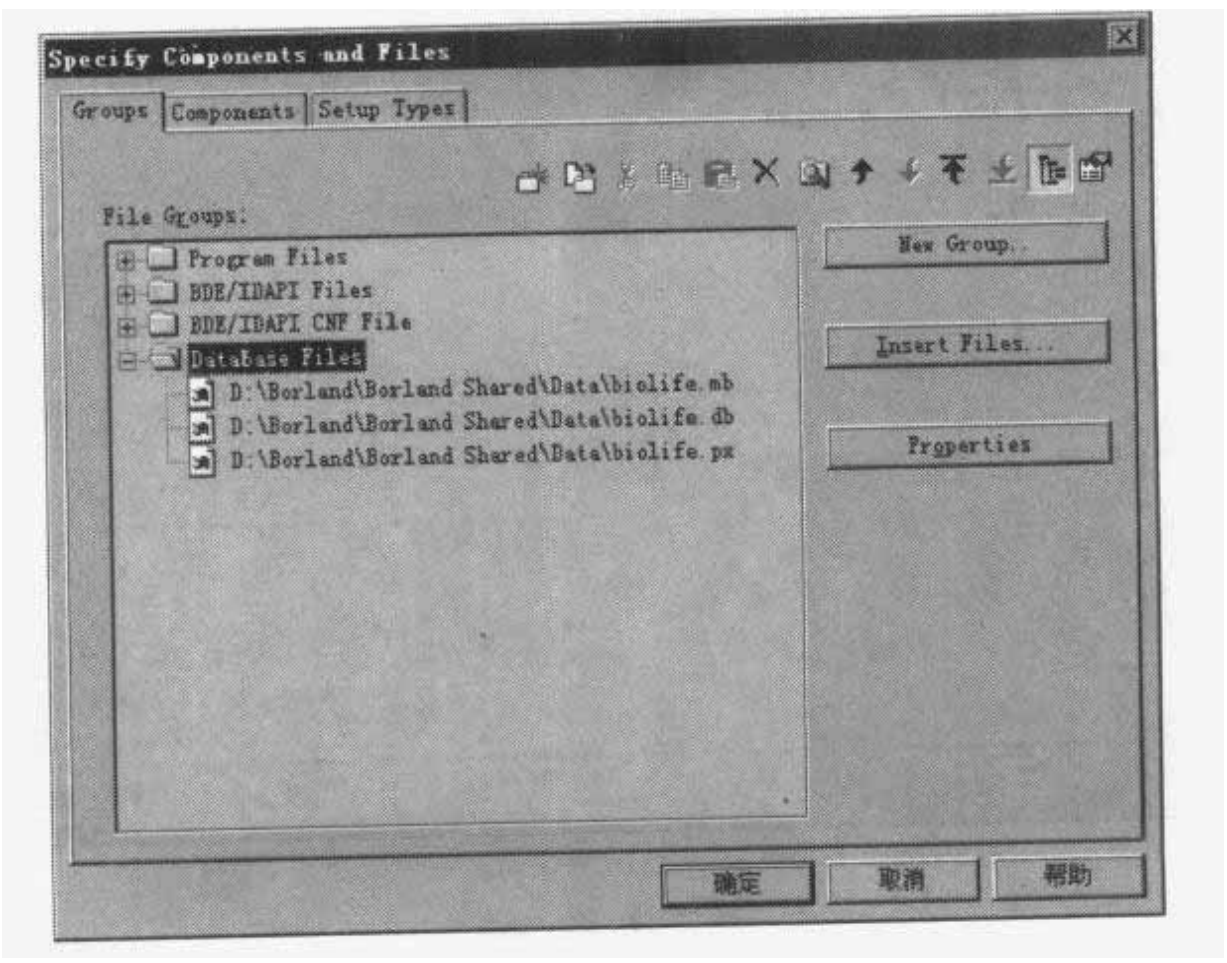


图 22-17　加入文件至 Database Files 群组

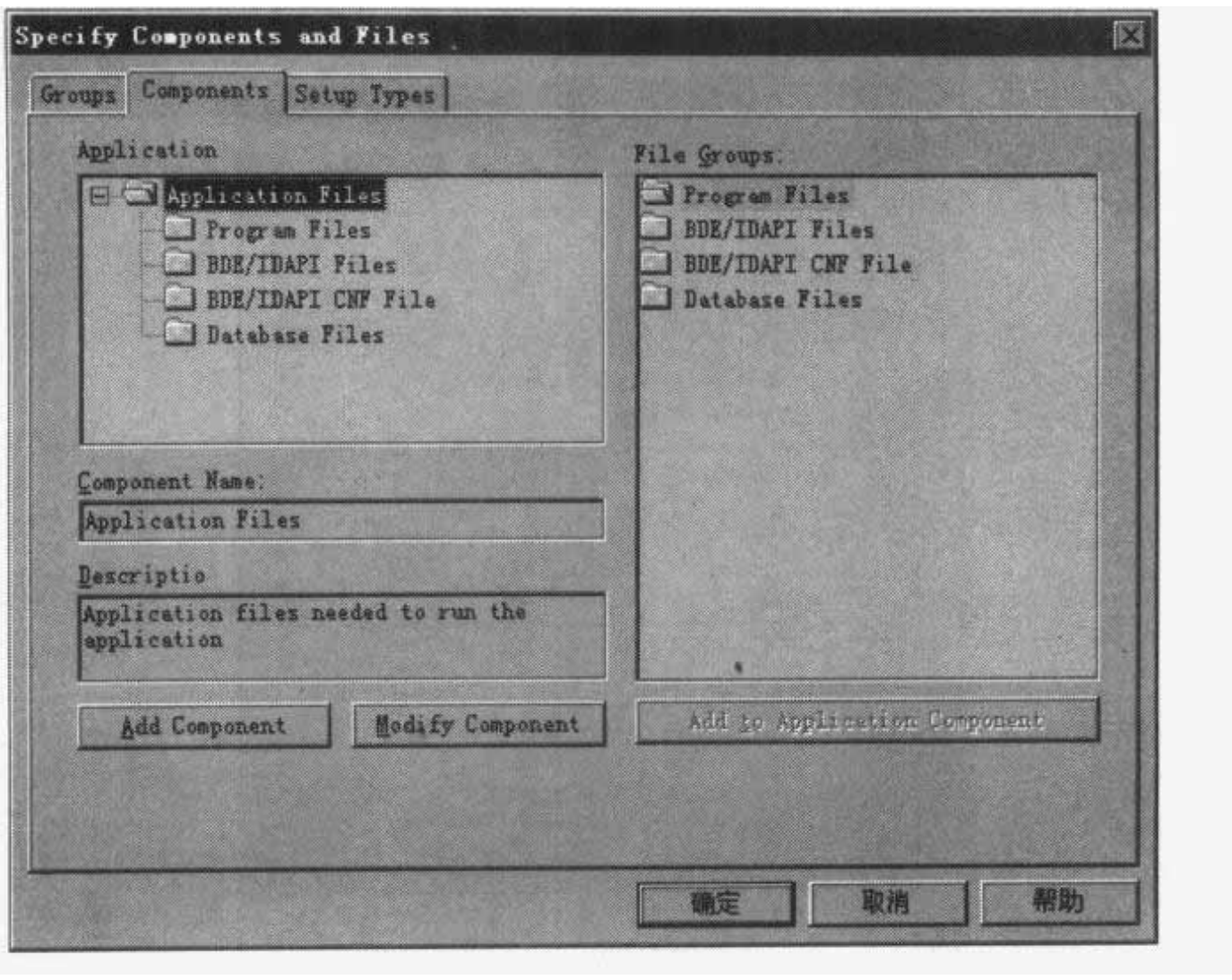


图 22-18　组件

组件：“Application Files”，如图 22-18 所示。

设定类型：如图 22-19 所示，本例中没有使用自定义类型，因此只有完整安装（Complete）类型。

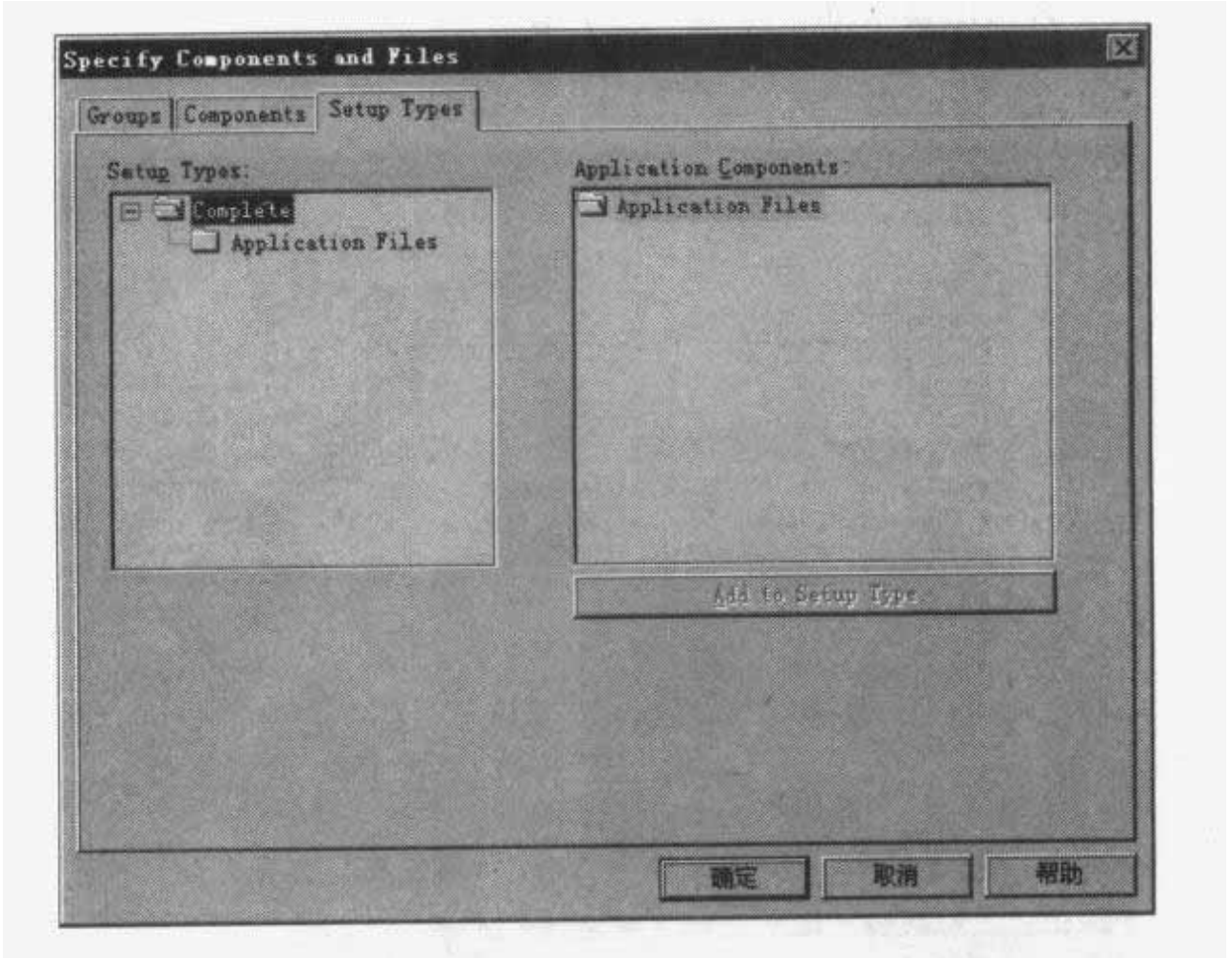


图 22-19　设定类型

（4）选择使用者界面元件

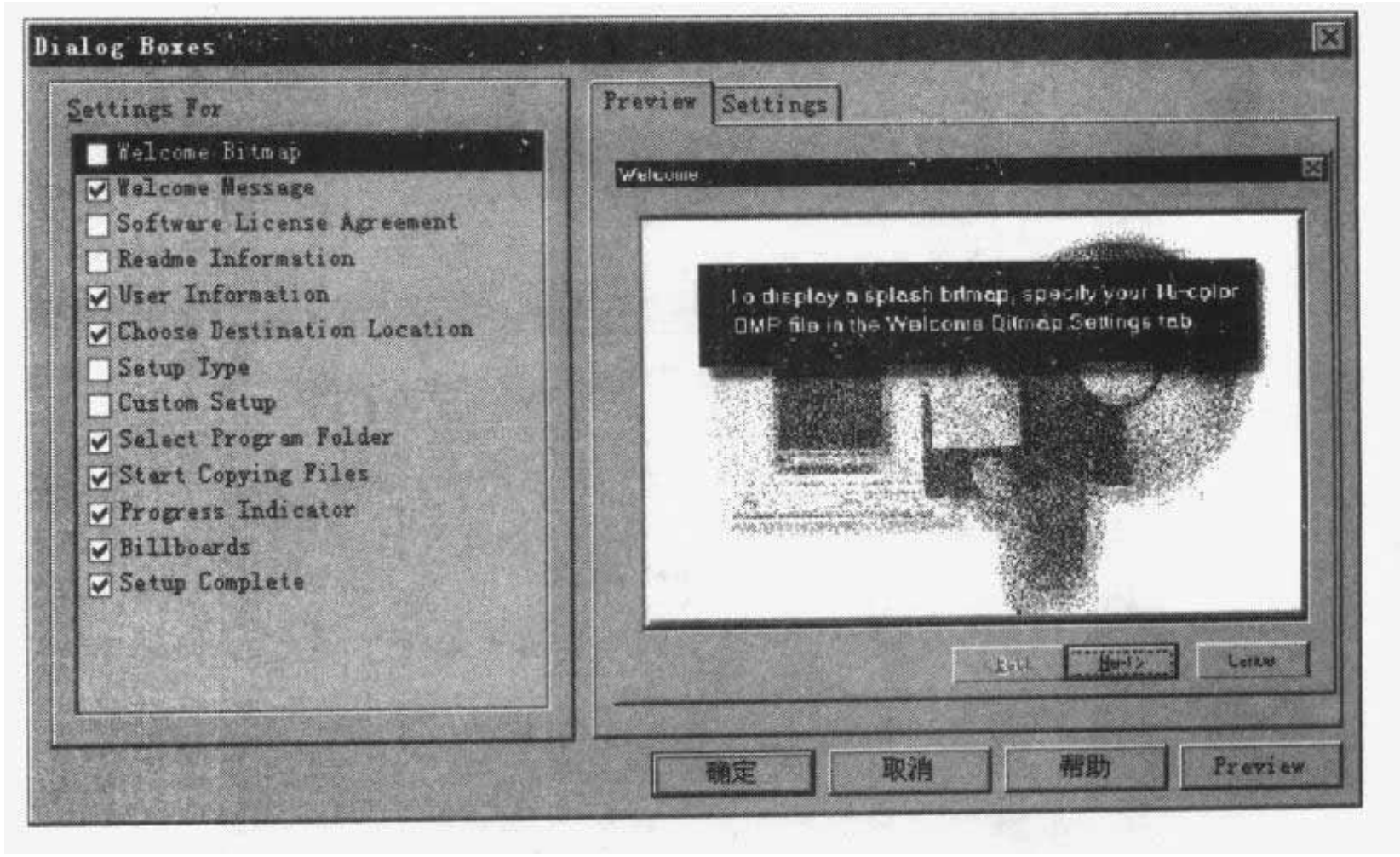


图 22-20　对话盒

如图 22-20 所示的对话盒主要用来设定在安装过程中所出现的相关信息视窗。

• Welcome Bitmap（欢迎 Bitmap）：可以设定自己的欢迎光临位图；

• Welcome Message（欢迎信息）：已有现成的欢迎信息，不选择欢迎消息则不会出现欢迎画面；

• Software License Agreement（软件授权协议）：软件使用的授权协议，系统有内定值，也可以自己编写；

• Readme Information（Readme 信息）：软件的说明信息或最新信息，一般均写入了 Readme.txt；

• User information （使用者信息）：安装本程序的使用者信息，例如姓名、公司名称以及序列号等；

• Select destination Location（选择目的地位置）：由使用者选择程序的安装目录；

• Setup type（设定安装类型）：本例选完全安装，所以不必选；

• Select Program Folder（选择程序文件夹）：设定安装的程序文件夹名称，预设为 Install Project 的名字；

• Start Copying Files（开始复制文件）：在开始复制文件前的设定信息，内容为设定的类型、目的地位置、使用者信息等；

• Progress Indicator（进程指示）：安装时是否出现安装百分比指示；

• Billboards：如果希望在安装时背景会有一连串的图形展示则须设置此项。设定此项时，要注意展示的图形文件名必须从 SETUP1.bmp 或 SETUP1.wmf 开始，然后依次编名；

• Setup Complete（安装完成）：当安装完成后，是否要安装程序启动应用程序或 readme，都可在此设定。

（5）变更注册表

本例未变更注册表，故省略。

（6）指定文件夹及图标

最后一个设定是指定程序安装的文件夹及代表程序的图标如图 22-21 和图 22-22 所示。

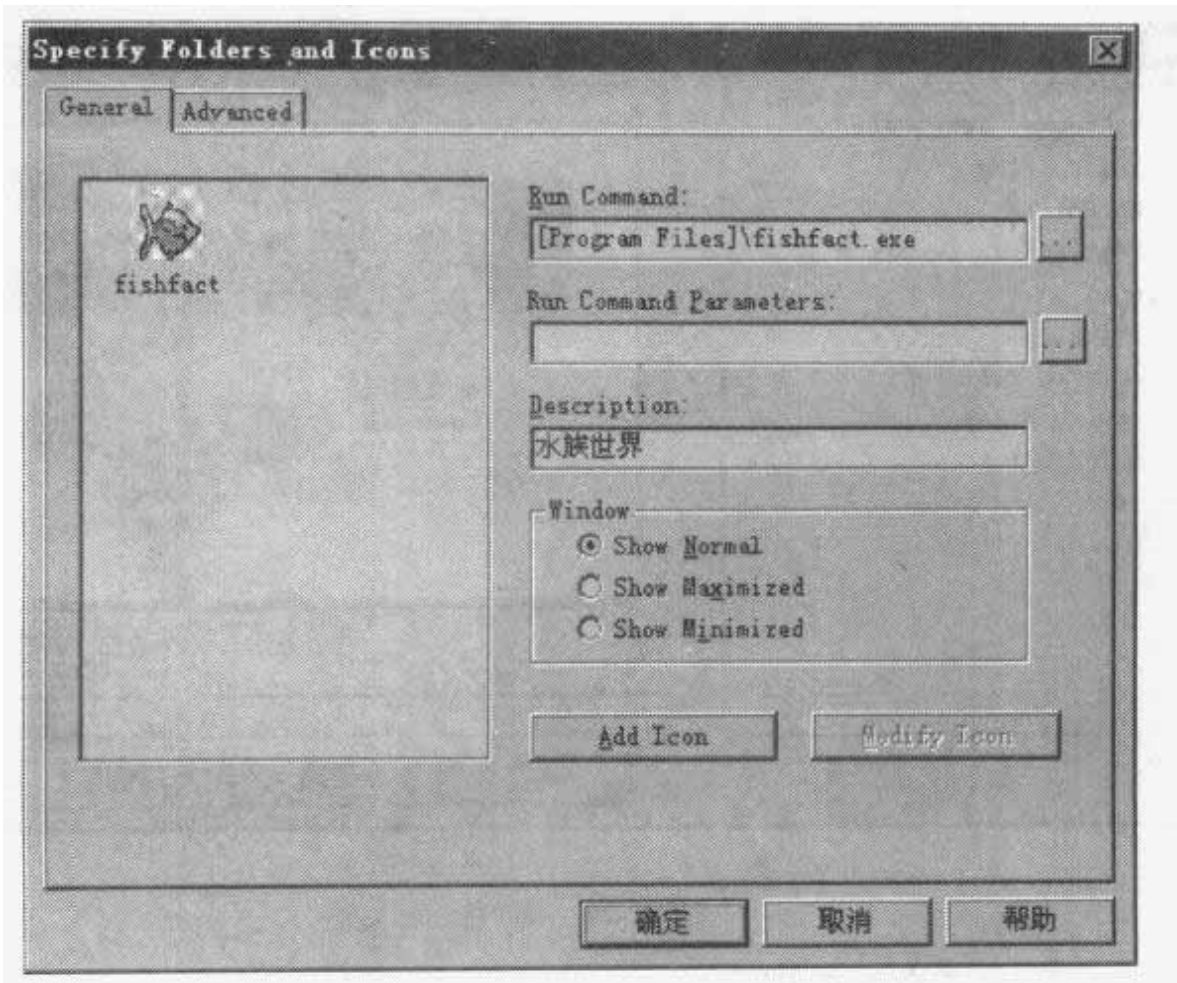


图 22-21　设定执行命令及参数

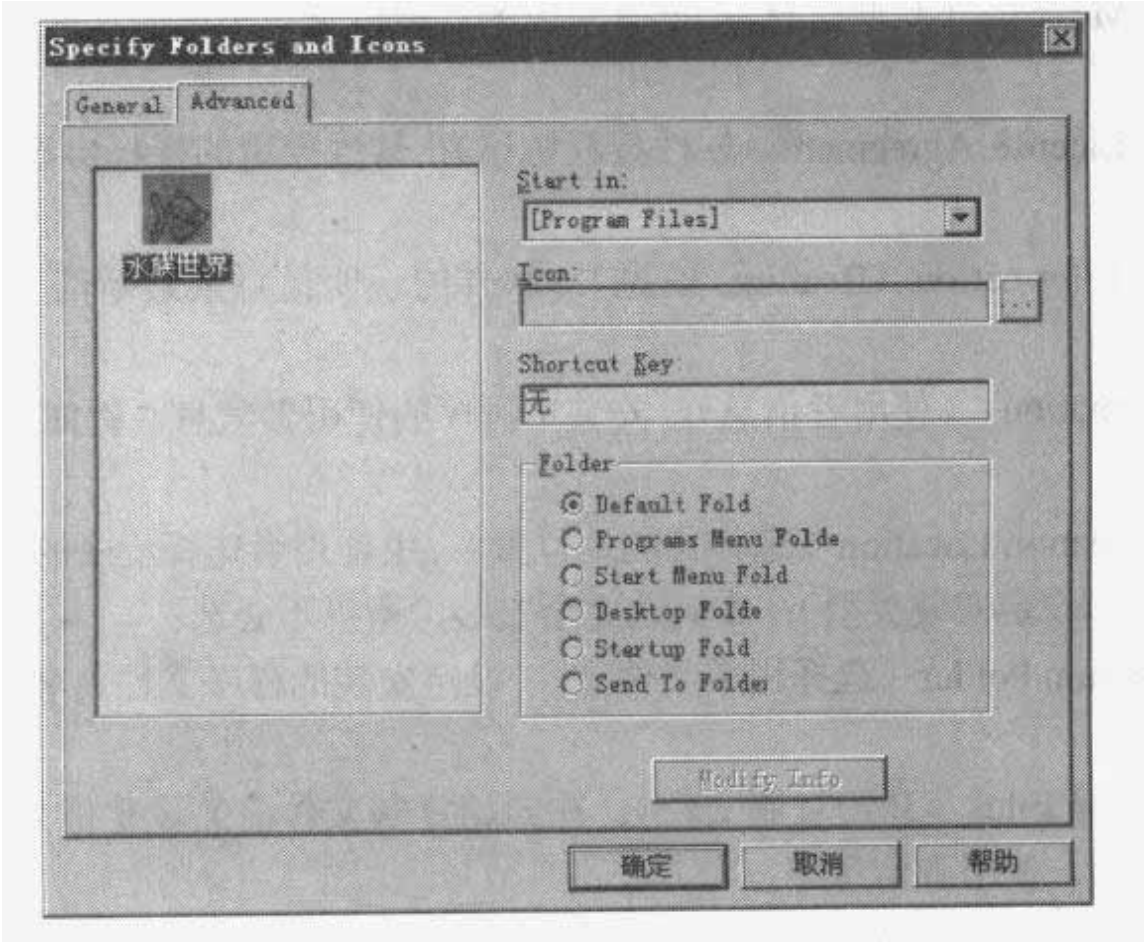


图 22-22　设定图标及工作路径

（7）执行磁盘建立工具

经过上述的设定，最后“Disk Builder”会自动将此设定压缩成自动安装磁盘，一套专业的安装软件就生成了。“Disk Builder”如图 22-23 所示，然后按“Build”按钮，完成后如图 22-24。

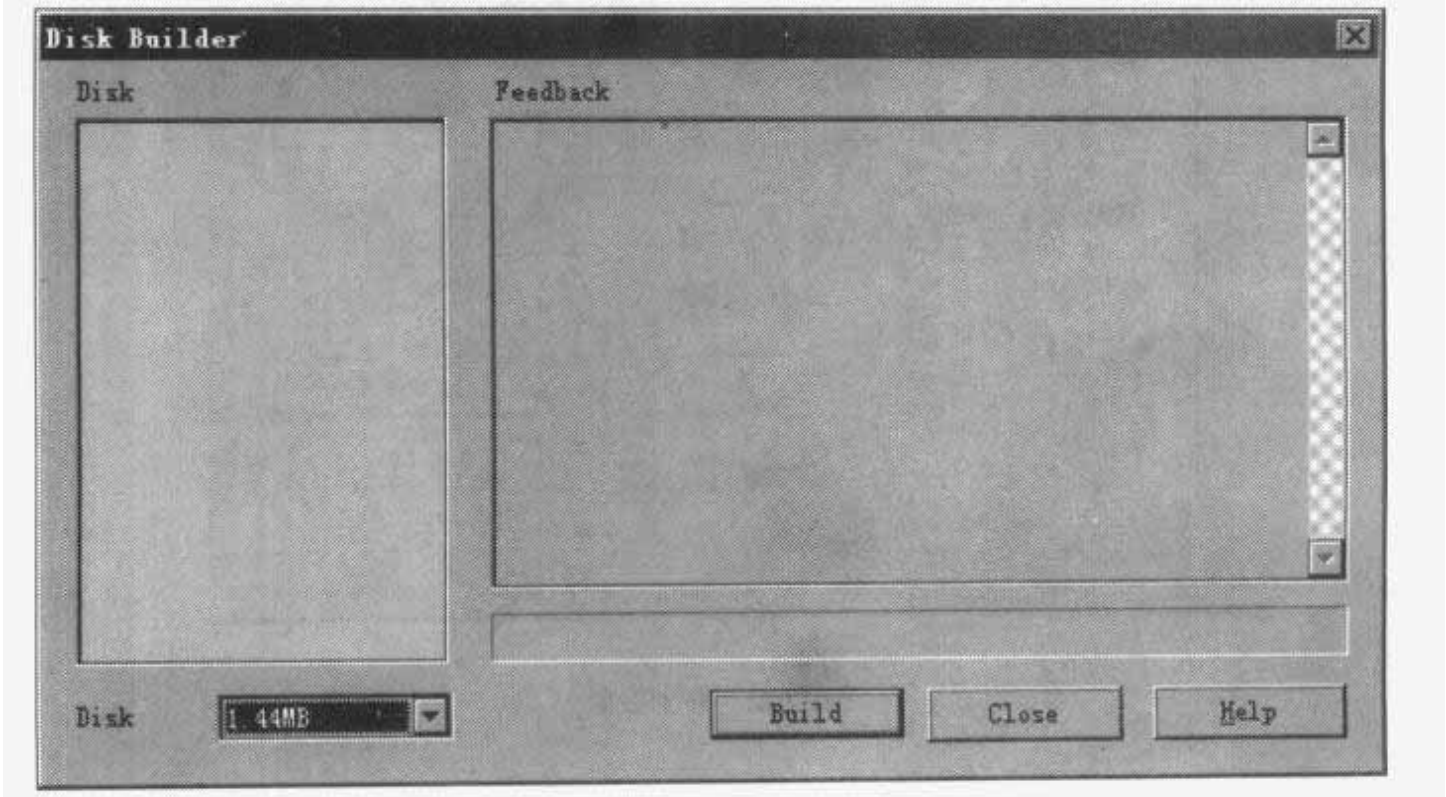


图 22-23　Disk Builder

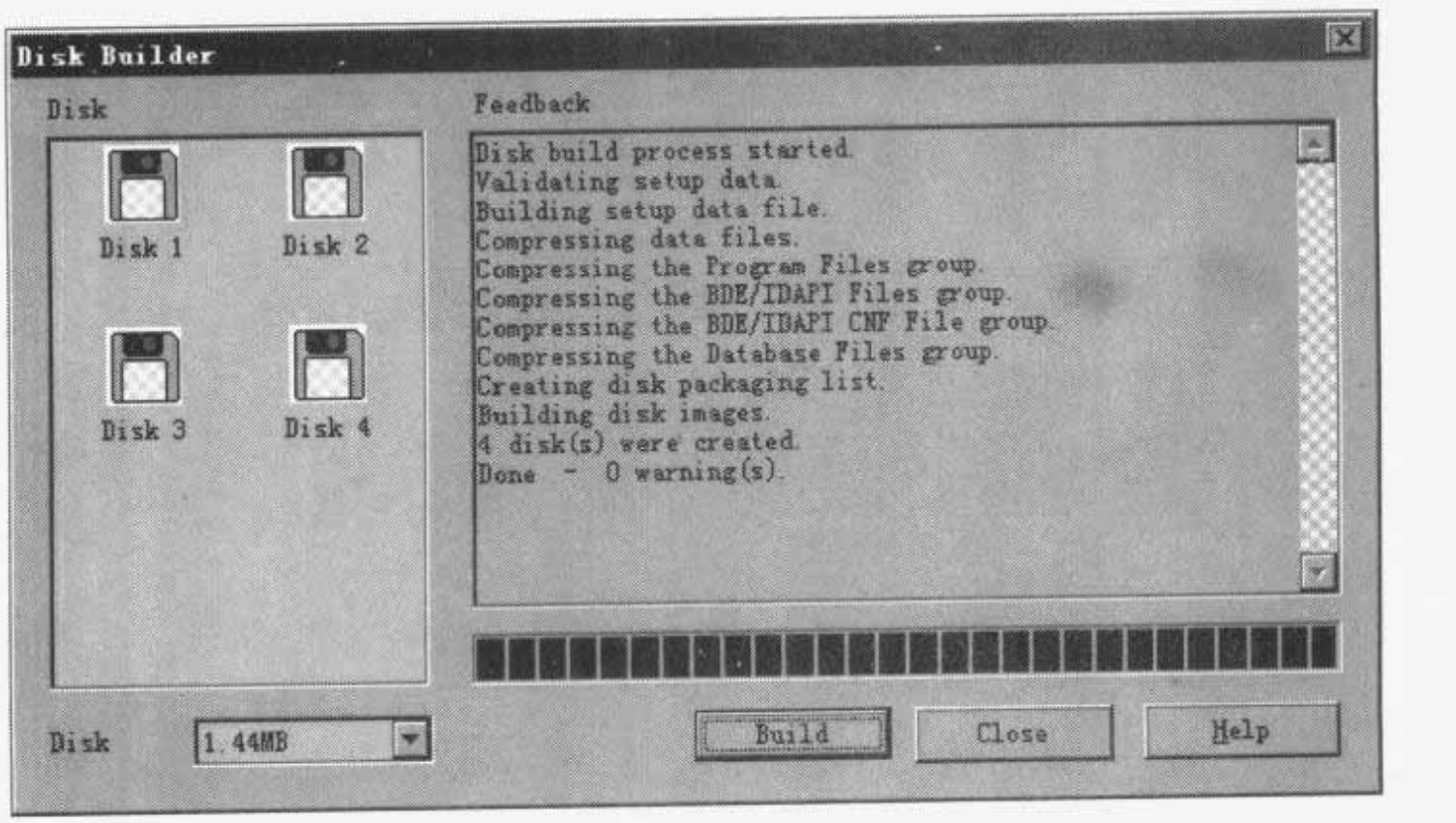


图 22-24　产生安装磁盘

建立磁盘若没有错误，系统会提示整个安装盘共需几片磁盘（系统依次产生 Disk1, Disk2, ..等目录）。

（8）复制至磁盘

将安装盘由硬盘中依次复制到磁盘（ A: ）如图 22-25 所示，然后就可以提供给客户安装或交由厂商压制成光碟包装出售。

看完上述的设定感觉如何？制作安装软件并不是遥不可及吧！